SMART ARM-based Microcontrollers

AT17629: SAM E70/V70 Ethernet Bootloader

APPLICATION NOTE

Introduction

The Atmel® | SMART ARM® Cortex®-M7 based MCUs deliver the highest performing Cortex-M7 based MCUs to the market with an exceptional memory and connectivity options for design flexibility making them ideal for the automotive, IoT, and industrial connectivity markets. The ARM Cortex-M7 architecture enhances performance and at the same time keeps the cost and power consumption under control.

Microcontrollers are used in a variety of electronic products. The devices are becoming more flexible due to the reprogrammable memory (Flash memory) often used to store the firmware of the product. This enables the firmware in a device to be upgraded in the field for correcting bugs or adding new functionalities.

Features

This application note provides an Ethernet-based bootloader implementation for Atmel SAM V7/E7 ARM Cortex-M7 based microcontrollers. It also discusses the boot sequence, upgrade sequence, safety, and security in bootloaders.

Table of Contents

1. Abbreviations

AES	Advanced Encryption Standard
ASF	Atmel Software Framework
CBC	Cipher Block Chaining
CFB	Cipher FeedBack
CTR	Counter
ECB	Electronic CodeBook
ICM	Integrity Check Monitor
MCU	Memory Control Unit
OFB	Output FeedBack
PC	Personal Computer
POR	Power On Reset
RAM	Random Access Memory
SHA	Secure Hash Algorithm
SP	Stack Pointer
VTOR	Vector Table Offset Register

2. Bootloader

Modern microcontrollers use Flash memory to store their application code. The main advantage of Flash is that the memory can be modified by the software itself. This is the key to in-field programming: a small section of code is added to the main application to provide the ability to download updates, replacing the old firmware of the device. This code is called a bootloader, as its role is to load a new program at boot.

The bootloader implementation poses several challenges such as correctly remapping memories, effective resource utilization, ensuring firmware upgrade is successful, and including necessary safety and security precautions.

A bootloader always resides in the memory to enable the firmware of the device to be upgraded at any time. Therefore, the bootloader must be as small as possible as it does not add any direct functionality for the user.

Downloading a new firmware into the device requires a process to initiate the bootloader to prepare for the transfer. There are two types of such trigger conditions: hardware and software.

A hardware condition can be triggered using a button press during a reset. Whereas, a software condition could be lack of a valid application in the system. When the system starts, the bootloader checks the predefined conditions. If one of them is true, it will connect to a host and wait for a new firmware. This host can be any device. However, a standard PC with the appropriate software is most often used. The firmware can be transferred using any protocol supported by the target such as RS232, USB, and CAN.

2.1. Boot Sequence

The start-up sequence of the Bootloader is as follows:

1. Initialization.
2. Trigger condition check.
3. Firmware upgrade (if trigger condition is set).
4. Firmware verification (optional).
5. Firmware loading (if verification is OK or disabled).

2.2. Upgrade Sequence

The basic upgrade flow starts with the host sending the firmware to the target, which then programs it into the memory. When the programming is complete, the new application is loaded. In theory, the “download” and “programming” steps are different, this is not the case in practice. Indeed, AT91SAM microcontrollers usually have 4x more Flash memory than RAM. The device cannot store the entire firmware in the RAM before writing it permanently to the Flash.

It is preferred that the code must be written to the memory while it is received. Since a Flash write operation consumes time, a communication protocol is required to halt the transfer when the data is being written and resume it afterwards.

The Flash memory is split up into fixed-size blocks called pages. Depending on the quantity of Flash in the microcontroller, the page size varies. A memory write operation can upgrade one page (or less) at a time; thus it is more logical to send packets containing one full page.

There are several optional post-processing features to consider. If code encryption is activated, then each page must be decrypted before being any data is written. If a digital signature or a message authentication code is available, it must be verified as soon as the download is complete.

2.3. Safety

It is important that a working firmware is embedded in the device at all times. However, the use of a bootloader can result in the conflicting situation where the new firmware has not been installed properly, compromising the behavior of the system.

This application note offers an example safety solution. Most of the techniques to circumvent those problems present a trade-off between the level of security and safety against the size and speed of the system. The safest and most secure solution is also probably the biggest and slowest in terms of performance. This also means that one must first carefully analyze safety and security requirements in a system, to implement only the required functionality.

Among the other features, a protocol stack is used in most communication standards to offer reliable transfers. This reliability is important for a bootloader application as the firmware must not get corrupted during the download.

2.4. Security

Securing a system enforces several features such as privacy, integrity, and authenticity.

2.4.1. Privacy

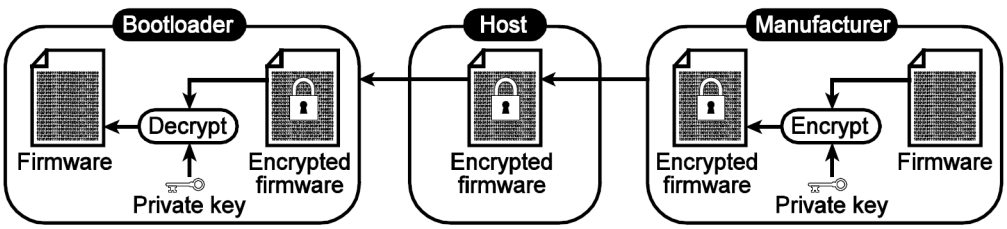
Privacy feature ensures that a piece of data cannot be accessed by unauthorized users or devices. It is a major concern for firmware developers to ensure that their application design is not accessible to the competitors. This feature ensures that the code is private and the target devices being the only authorized “users”.

Microcontrollers provide a mechanism making it difficult for malicious users to read the program code written in the device. However, for in-field firmware upgrade, the manufacturer has to provide the new code image to customers so they can patch their devices themselves. Which on the other hand enables a skilled person to potentially decompile and retrieve the original code.

Data privacy is enforced using encryption: the data is processed using a cryptographic algorithm along with an encryption key, generating a cipher text which is different from the original one. Without the specific decryption key, the data will be illegible, preventing anyone unauthorized from reading it.

A private-key algorithm is used to generate the encrypted firmware. A public-key system cannot be used, as the firmware would then be decipherable by anyone. The encryption and decryption keys are thus identical and shared only between the bootloader and the manufacturer.

Figure 2-1. Firmware Encryption



All security issues cannot be solved by encrypting the code. An attacker can probably pinpoint the location in the code of an important variable and tweak it until he gets the desired result.

The code encryption combines itself with a message authentication code. Since they both use a symmetric encryption algorithm, one can use the same algorithm to save the code size.

2.4.2. Selecting an Algorithm

A symmetric encryption algorithm can be defined by several characteristics:

- Key length in bits
- Block length in bits
- Security
- Size and Speed

The *Key length* used by an algorithm is an important parameter of the algorithm. The larger it is, the more difficult it is to perform a brute-force attack. As computers become faster and faster, longer keys are required. A reasonable length seems to be 128 bits at the moment, it is one of the Key lengths selected for Advanced Encryption Standard (AES) cipher.

A large *Block length* is required to avoid a “code-book attack”, i.e., someone getting enough blocks of plain text and their corresponding cipher text to build a table, enabling decipher further information. In this scenario (firmware upgrading), an attacker is very unlikely to get access to any plain text at all. Therefore, the block length can be of any reasonable size. Most block ciphers will use at least 64 bits, with modern ciphers using at least 128 bits.

2.4.3. Modes of Operation

Different modes of operations are possible when using a symmetric cipher:

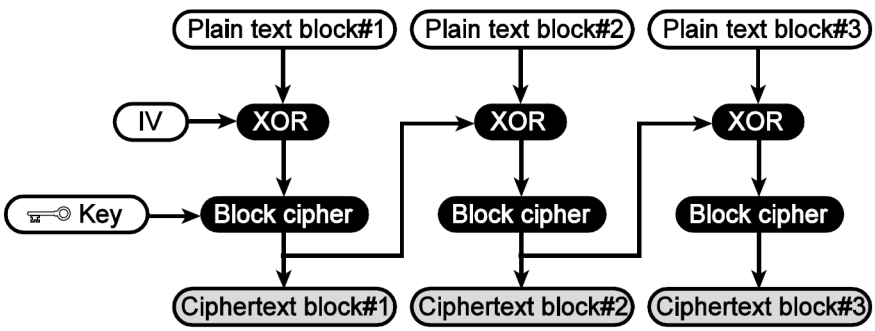
- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC, CFB, OFB, CTR)
- Authenticated Encryption (EAX, CCM, OCB)

The basic mode of operation is *ECB*. In this mode each block of plain text is encrypted using the key and the selected algorithm, resulting in a block of cipher text. However, this mode is very insecure as it does not hide the patterns. Two identical blocks of plain text will be encrypted to the same cipher text block.

To solve this issue, *Cipher Block Chaining* modes are used. Encryption is not only done with the current block of plain text, but also with the last encrypted block. It makes each block depend on the previously encrypted data, making everything interdependent.

The first block is encrypted using a random *Initialization Vector* (IV). While this vector can be transmitted in clear text, the same vector should not be reused with the same key. A manufacturer will produce more than one firmware upgrade for a product in its lifetime. So, the IV cannot be stored in the chip similar to the key. It has to be transmitted by the host.

Figure 2-2. CBC Mode of Operation



2.4.4. Integrity

Integrity feature ensures that any change in the data is detected. For example, an authorized firmware may be slightly modified. Although the firmware may appear genuine, there could be attacks which might change the data.

To verify Integrity, check for:

- an intentional modification of the firmware
- an accidental modification of the firmware

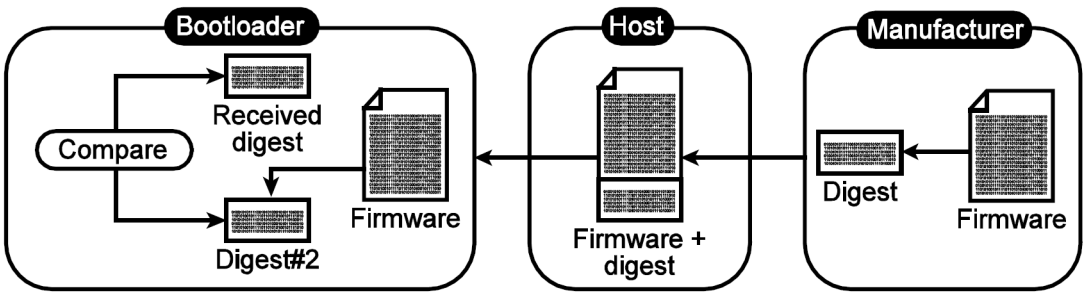
Accidental modification is a safety problem. It is typically solved by using error detection codes.

Hash Function produces a digital “fingerprint” for a piece of data. It means conversely to an error detection code, every piece of data must have its own unique fingerprint.

To verify the integrity of a firmware, its fingerprint is calculated and attached to the file. When the bootloader receives both the firmware and its fingerprint, it re-computes the fingerprint and compares it to the original fingerprint. If both are identical, then the firmware has not been altered.

In practice, a hash function takes a string of any length as an input and produces a fixed-size output called a message digest. It also has several important properties such as good diffusion (the ability to produce a completely different output even if only one bit of the input is flipped).

Figure 2-3. Firmware Hashing



Since the output length is fixed irrespective of the input, it is not possible to generate a unique digest for every piece of data. However, hash functions ensure that it is almost impossible to find two different messages with the same digest. This achieves almost the same result as uniqueness, at least in practice.

The disadvantage of hashing the firmware is its simplicity. An attacker can easily modify the file and re-compute the hash. The bootloader would not be able to detect that modification.

During run time, a hash function can be used to verify the firmware integrity to avoid executing a corrupted application.

2.4.5. Hash Functions

A hash function has three defining characteristics:

1. Output length.
2. Security.
3. Size and Speed.

The output length of a hash must be large enough to make it almost impossible to find collisions (two different files having the same digest). This prevents anyone from finding a piece of data producing the same hash. Most modern hash functions has at least a 160-bit output (like SHA-1).
Note: Of late the hash algorithms with hash length of 512 bits are preferred.

The security of the hash function is much more critical than the length of its output. Indeed, MD5 (which only has a 128-bit output) would still be secure if it did not have serious design flaws in it. Similarly to block ciphers, finding a flaw in a hash does not imply that it has been cracked; most attacks are not feasible without gigantic computational power. Recent designs considered secure should be preferred rather than using any deprecated algorithms.

When deciding on a hash function, its size and speed performances should also be evaluated. However, the stronger algorithms are often the slowest ones (which is not true for block ciphers), so there will be a security/speed trade-off.

The most commonly used hash algorithms are listed in the following table.

Table 2-1. Hash Algorithms

Algorithm	Hash length	Security	Size and speed
MD5	128 bits	Broken	Fast
RIPEMD-160	160 bits	Secure	Slow
SHA-1	160 bits	Broken	Slow
SHA-256	256 bits	Secure	Slow
WHIRLPOOL	512 bits	Secure	Very Slow
Tiger	192 bits	Secure	Fast
HAVAL	128 to 256 bits	Broken	Moderately Fast

2.4.6. Authenticity

Authenticity feature helps to verify if the firmware is from the authentic manufacturer. While reprogramming a device, the firmware could be from a third party. This may be problematic if that firmware is intended for malicious use such as bypassing security protections and illegally using critical functions of the device.

Authentication is about verifying the identity of the sender and the receiver of a message. A bootloader should verify authenticity of the manufacturer and target.

3. TFTP Protocol

TFTP is a simple protocol to transfer files, and therefore it was named the Trivial File Transfer Protocol or TFTP. It has been implemented on top of the Internet User Datagram Protocol (UDP or Datagram). It is designed to be small and easy to implement. Therefore, it lacks most of the features of a regular FTP. The only thing it can do is read and write files (or mail) from/to a remote server. It cannot list directories and currently it has no provisions for user authentication. In common with other Internet protocols, it passes 8-bit data bytes.

Any transfer begins with a request to read or write a file, which also serves to request a connection. If the server grants the request, the connection is opened and the file is sent in fixed length blocks of 512 bytes. Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet before the next packet can be sent. A data packet of less than 512 bytes signals termination of a transfer.

Note: Both machines involved in a transfer are considered senders and receivers. One sends data and receives acknowledgments while the other sends acknowledgments and receives data.

An error is signaled by sending an error packet. This packet is not acknowledged, and not retransmitted (i.e., a TFTP server or user may terminate after sending an error message), so the other end of the connection may not get it.

TFTP uses UDP as its transport protocol. A transfer request is always initiated targeting port 69, but the data transfer ports are chosen independently by the sender and receiver during the transfer initialization. The ports are chosen at random according to the parameters of the networking stack.

3.1. TFTP Packets

TFTP supports five types of packets, all of which have been mentioned in following table:

Table 3-1. TFTP Opcodes

Opcode	Operation
1	RRQ - Read Request
2	WRQ - Write Request
3	DATA - Data
4	ACK - Acknowledgment
5	ERROR - Error

Table 3-2. RRQ Format

Opcode=1 (2 bytes)	**File Name** (Variable length)	**All 0s** (1 byte)	**Mode** (Variable length)	**All 0s** (1 byte)

Table 3-3. WRQ Format

Opcode=2 (2 bytes)	**File Name** (Variable length)	**All 0s** (1 byte)	**Mode** (Variable length)	**All 0s** (1 byte)

Table 3-4. DATA Format

Opcode=3 (2 bytes)	**Block Number** (2 bytes)	**Data** (0 - 512 bytes)

Table 3-5. ACK Format

Opcode=4 (2 bytes)	**Block Number** (2 bytes)

Table 3-6. ERROR Format

Opcode=5 (2 bytes)	**Error Number** (2 bytes)	**Error Data** (Variable length)	**All 0s** (1 byte)

The File Name is a sequence of bytes in ASCII terminated by a zero byte.

The Mode field contains the string "netascii", "octet", or "mail" in ASCII indicating the three modes defined in the protocol. A receiver, which receives netascii mode data must translate the data to its own format. Octet mode is used to transfer a file that is in the 8-bit format of the machine from which the file is being transferred. Mail mode uses the name of a mail recipient in place of a file and must begin with a WRQ.

The WRQ and DATA packets are acknowledged by ACK or ERROR packets, while RRQ and ACK packets are acknowledged by DATA or ERROR packets. The block number in an ACK echoes the block number of the DATA packet being acknowledged. A WRQ is acknowledged with an ACK packet having a block number of zero.

An ERROR packet can be the acknowledgment of any other type of packet. The error code is an integer indicating the nature of the error.

The TFTP header of a packet contains the opcode associated with that packet.

3.2. TFTP Option Extensions

This extension allows file transfer options to be negotiated prior to the transfer using a mechanism, which is consistent with TFTP's Request Packet format. TFTP options are appended to the Read Request and Write Request packets. A new type of TFTP packet, the Option Acknowledgment (OACK), is used to acknowledge a client's option negotiation request. Options are appended to a TFTP Read Request or Write Request packet. The following tables illustrates RRQ and WRQ packet frames with Options.

Table 3-7. RRQ Format

Opcode=1 (2 bytes)	**File Name** (Variable length)	**All 0s** (1 byte)	**Mode** (Variable length)	**All 0s** (1 byte)	**Opt1** (Variable length)	**All 0s** (1 byte)	**Value1** (Variable length)	**All 0s** (1 byte)

Table 3-8. WRQ Format

Opcode=2 (2 bytes)	**File Name** (Variable length)	**All 0s** (1 byte)	**Mode** (Variable length)	**All 0s** (1 byte)	**Opt1** (Variable length)	**All 0s** (1 byte)	**Value1** (Variable length)	**All 0s** (1 byte)

Table 3-9. OACK Format

Opcode=6 (2 bytes)	**Opt1** (Variable length)	**All 0s** (1 byte)	**Value1** (Variable length)	**All 0s** (1 byte)

Note: Opt1 and Value1 are used only for representation. Actual options and values used in this app note are detailed below.

The options and values are all NULL-terminated, in keeping with the original request format. If multiple options are to be negotiated, they are appended to each other. The order in which options are specified is not significant.

3.3. TFTP Option Negotiation

The client appends options at the end of the Read Request or Write Request packet. If the server supports option negotiation, and it recognizes one or more of the options specified in the request packet, the server responds with an Options Acknowledgment (OACK). Each option the server recognizes and accepts the value for is included in the OACK.

An option not acknowledged by the server must be ignored by the client and server as if it were never requested. If multiple options were requested, the client must use those options which were acknowledged by the server and must not use those options which were not acknowledged by the server.

Following are some of the options defined in the Option Extensions and used in this application note.

TFTP Option – blksize

The TFTP Read Request or Write Request packet is modified to include the block size option. The “blksize” is a NULL terminated options followed by number of octets in a block, specified in ASCII. This is a NULL terminated field.

If the server is willing to accept the blksize option, it sends an Option Acknowledgment (OACK) to the client. The specified value must be less than or equal to the value specified by the client. The client must then either use the size specified in the OACK, or send an ERROR packet to terminate the transfer.

TFTP Option – tsize

The TFTP Read Request or Write Request packet can be modified to include the tsize option. The “tsize” is a NULL terminated option followed by file size in octets, specified in ASCII. This is a NULL terminated field.

In Read Request packets, a size of "0" is specified in the request, then the size of the file, in octets, is returned in the OACK. If the file is too large for the client to handle, it may abort the transfer with an Error packet.

In Write Request packets, the size of the file, in octets, is specified in the request and echoed back in the OACK. If the file is too large for the server to handle, it may abort the transfer with an Error packet.

TFTP Option – timeout

The TFTP Read Request or Write Request packet can be modified to include the timeout option. The “timeout” is a NULL terminated option followed by the number of seconds to wait before retransmitting, specified in ASCII. Valid values range between "1" and "255" seconds. This is a NULL-terminated field.

4. Example Bootloader and User Application

4.1. Hardware/Software Requirements

- Hardware Prerequisites
  - Atmel SMART SAM E70 Xplained ULTRA Kit
  - Interfacing Cables
    - One Micro-USB type-B cable (EDBG)
    - CAT V Ethernet cable
- Software Prerequisites
  - Atmel Studio 7.0
  - ASF 3.32.0
  - tftpd32.exe or tftpd64.exe
  - DHCP server running in the network

4.2. Design Considerations

4.2.1. Safety

- Erase operations are triggered only on the user application area. This ensures that the bootloader is always available to run on the device.
- TFTP protocol is chosen for data transfers between Server and Client. This takes care of Error detection, Transmission loses, and Packet acknowledgment.

4.2.2. Security

- AES Cipher Block Chaining (CBC) algorithm is implemented to ensure data privacy. This encryption is not only done with the current block of plain text, but also with the last encrypted block. Before sending data to the device, it is encrypted page by page with predefined Key and Initial Vector.
- SHA-1 algorithm is used to produce digital “fingerprint“ for the entire user application. Before executing the user application, the bootloader verifies fingerprint to check integrity of the user application.

4.2.3. Software Considerations

- This design uses AES and ICM hardware modules for Decryption, Hash functions, and Software Components, Services provided in ASF
- Applications Start address and length are aligned to Page boundaries, hardware requirements such as Vector Table locations, and alignment for SHA padding requirements
- Other than 32 bytes used for applications switching, the entire RAM is available for both bootloader and user applications
- A soft reset is required to jump to the user application after completing the Programming sequence
- A soft reset is required to switch to bootloader after receiving switch to bootloader command/ sequence
- A PC tool is required to:
  - Post process user application binaries to populate SHA values in the binary files and Encrypt if required
  - Create a configuration file based on information available in user application binary files

- The current configuration file contains the user application start address, size, application version, AES Key, and Initial Vectors

4.2.4. Hardware Requirements

In this example the following hardware points are considered on ATSAME70Q21:

1. Flash wait-states (six wait-states) are enabled as per device suggestions.

4.2.5. Software Limitations

1. For the first time it is required to program the bootloader with help of available programmers.
   1.1. If bootloader is erased accidentally it must be programed again using programmers/other available options.
2. If power cycle occurs between jump to bootloader instruction and board reset, the control is returned to the user application. In such case, it is required to issue jump to bootloader instruction once again.
3. Applications customized to download using bootloaders cannot run on its own. This because its start address is changed to a different location in the flash.

4.3. Software Architecture

4.3.1. Features

The features of the example implementation are:

- Basic boot loading capabilities over Ethernet using TFTP protocol to upgrade the firmware
- TFTP client is implemented using LWIP stack
- DHCP feature is enabled to get IP address assigned to XPLD board
- Code encryption using AES CBC
- Both bootloader and user applications are provided with Footers, which contains:
  - Application Versions
  - Applications Start Address and Length
  - Option to add Authentication information
  - SHA-1 padding bytes
  - Hash Digest
- Example user application that toggles LED and switches to bootloader on SW0 key press

4.3.2. Code Locations

The Flash is divided into two sections; one for bootloader and the other for user application. Each section reserves 256 bytes Footer at the end of the section. Following graphic illustrates the Flash Sections and Application Footer information.

Figure 4-1. Flash Sections

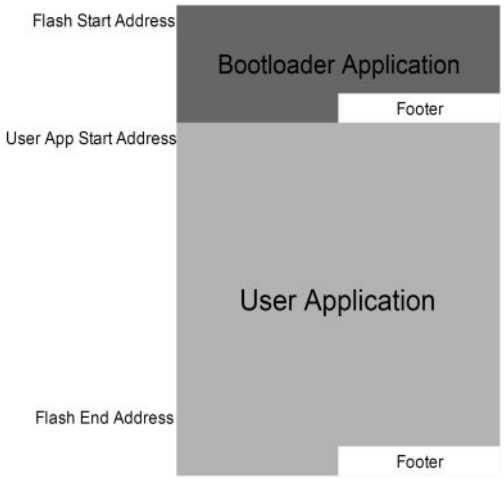


Table 4-1. Application Footer

Parameter	Data type	Description
App Version	`uint32_t`	Application Software's Version
App Jump Handler	`void (*fpJumpHandler) (void)`	Function pointer to Jump to application
Boot App Start Address	`uint32_t`	Start address of bootloader
Boot App Length	`uint32_t`	Length of bootloader
User App Start Address	`uint32_t`	Start address of user application
User App Length	`uint32_t`	Length of user application
Reserved	`uint32_t[24:0]`	Reserved. Used for application authentication, if required.
SHA Padding Data	`uint32_t[15:0]`	SHA algorithms requires padding bytes. This space is reserved for them.
SHA Digest	`uint32_t[15:0]`	SHA output is stored here. Bootloader uses this value to check Integrity of user application.

4.3.3. Switching Between Applications

4.3.3.1. Bootloader to User Application

In the bootloader the Jump Signature plays a critical role in deciding whether the control should remain in the bootloader or check for the user application execution.

Jump Signature is the RAM memory reserved by both bootloader and user application. "stay_in_bootloader" is a string, which will be loaded in to the Jump Signature location when the control must remain in the bootloader on the next soft reset.

- On POR it contains zeros, which makes the bootloader to check for user application existence
- If the user application requires control to remain in the bootloader, it updates the Jump Signature with "stay_in_bootloader"

- When the bootloader detects “stay_in_bootloader”, it clears the Jump Signature and remains in the bootloader until the next soft reset

On soft reset the control starts executing the bootloader. The bootloader:

1. Checks for Jump Signature if it is not “stay_in_bootloader”:
    1.1. Reads bootloader Footer to determine the user application Start address and Length.
    1.2. Reads the user application Footer with help of the Start Address and Length.
    1.3. Calculates digest for the user application and compares it with the digest available in the Footer.
    1.4. On detecting valid digest:
        1.4.1. Releases resources such as peripherals.
        1.4.2. Disables interrupts (if required).
        1.4.3. Updates vector table offset (VTOR) and Stack pointer (SP).
        1.4.4. Jumps to reset handler provided in the vector table.
2. If Jump Signature is “stay_in_bootloader”:
    2.1. Initialize Flash Wait states.
    2.2. Initiates communication with TFTP server and requests for configuration file.
    2.3. Validates received Configuration file.
    2.4. On detecting valid Configuration file:
        2.4.1. Erases the user application area.
        2.4.2. Requests for the user application image file and starts programming it.

The following flowcharts present Boot and Upgrade sequences used in this example.

Figure 4-2. Boot Sequence Diagram

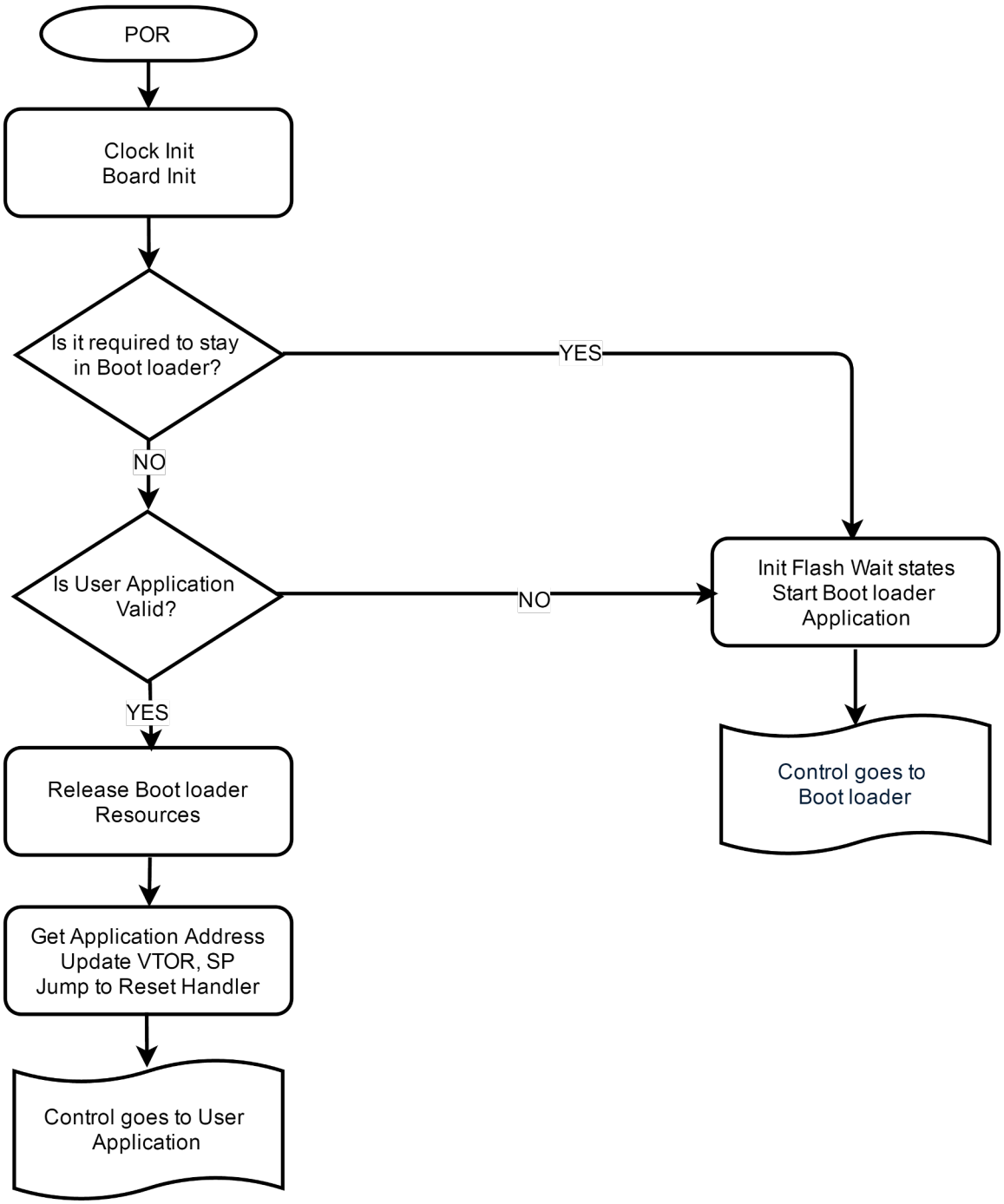
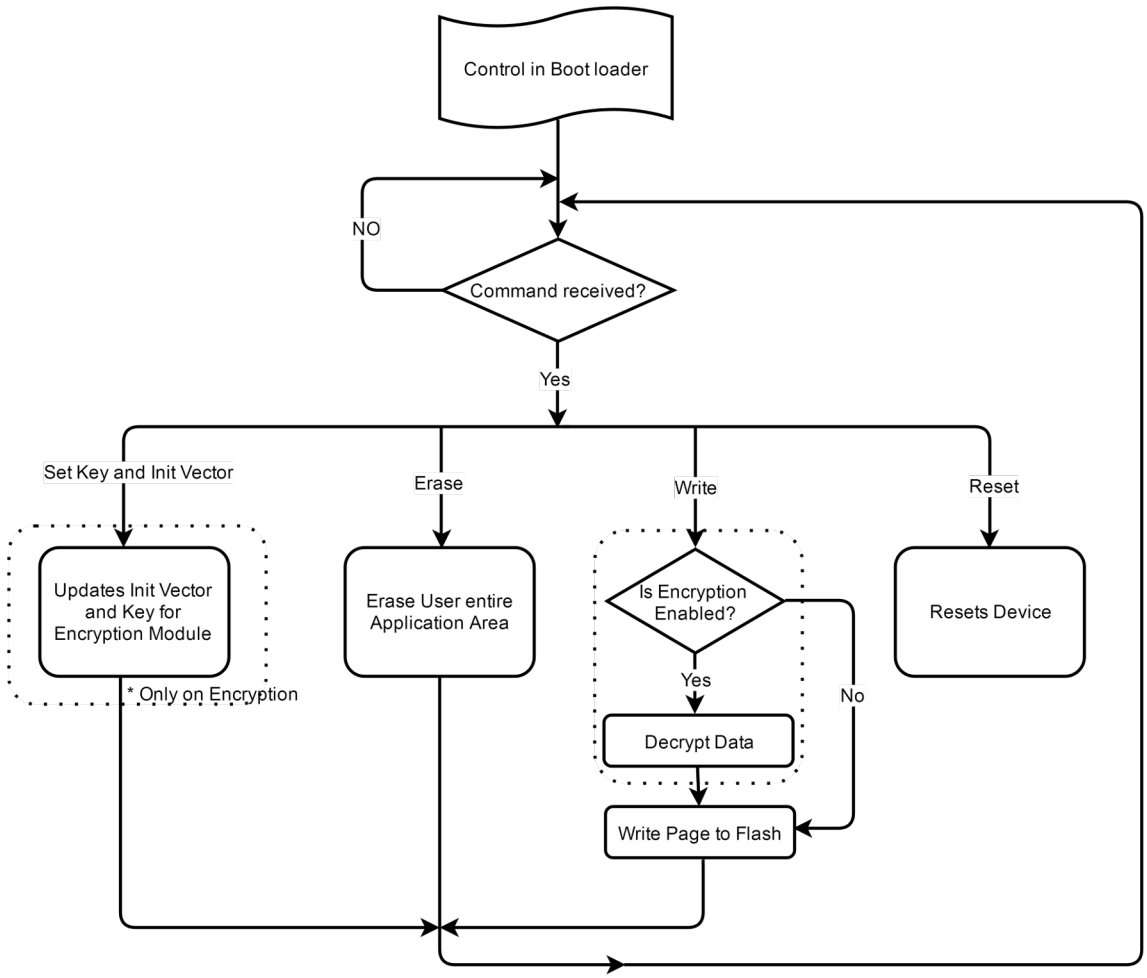


Figure 4-3. Firmware Upgrade Sequence



4.3.3.2. User Application to Bootloader

When the user application receives a request for firmware upgrade,

1. Reads user application Footer to know bootloader Start Address and Length.
2. Reads bootloader Footer for bootloader Jump handler and executes it.
   - Bootloader Jump handler updates Jump Signature with “stay_in_bootloader”.
3. At this point, a soft reset is required. On receiving soft reset the control goes to the bootloader and remain there as Jump signature indicates it.

4.3.3.3. Linking Options

The linker modifications are done for this example project. Linker file is updated to

- Define application’s start address and length in `rom` section
- Define Footer at the end of the application section
- Reserve 32 bytes of RAM (Jump Signature) at the start so that both applications use it
- Fill unused locations with known data (0xFF)

In this example, the bootloader and the user application are defined at the following locations:

Table 4-2. Memory Allocations

Application type	Section	Flash start address	Reserved space
Bootloader	Application	0x00400000	0x0000BF00
	Footer	0x0040BF00	0x00000100
User Application	Application	0x0040C000	0x001F3F00
	Footer	0x005FFF00	0x00000100

32 bytes of RAM (Jump Signature) is reserved so that space is available for both the bootloader and the user application to share the next steps to carry on.

Linker options for both the bootloader and the user applications are modified as follows:

- `rom (rx)` `ORIGIN` and `LENGTH` updated to match above
- Created `ApplicationFooter (rx)` section to hold footer information
- Created `JumpSignature (rwx)` section to reserve RAM
- Fill unused locations of the application with known data (0xFF)

4.3.3.4. User Application Customization for Bootloader

The following section provides detailed steps to convert a standard user application compatible with this bootloader. This process requires:

1. Remapping of flash locations.
2. Adding application Footers.
3. Reserving RAM location.
4. Fill unused locations with 0xFF.
5. Define sequence for bootloader switching.

Remapping of Flash Locations

It is important to avoid storing user application in the bootloader locations. As described in previous sections, the user application should be remapped to a different location. This can be achieved with help of linker file. For example:

```
rom (rx): ORIGIN = 0x0040C000, LENGTH = 0x001F3F00
```

This sets start address of the user application to the address specified as ORIGIN defined above and allocates specified LENGTH bytes for user application.

Ensure that the user application start address and the length are updated in Footers because bootloader reads this information before jumping to the user application.

Note: Ensure that the Flash information stored in the bootloader Footer and the user application Footer are the same.

Adding Application Footer

Application Footer is an important parameter used by both user application and bootloader. They use these parameters to read critical information before switching to other application. It is important to place this at the end of the Flash memory allocated to the user application.

- Adding Footer in Application

```
/** Application Footer information */
__attribute__ ((section(".application_footer_data")))
const ts_application_footer ms_user_application_footer =
{
    {
        U32_USER_APP_SW_RELEASE_VERSION,
```

```
        NULL,
        U32_BOOT_LOADER_APPLICATION_START_ADDRESS,
        U32_BOOT_LOADER_APPLICATION_ALLOCATED_SIZE,
        U32_USER_APPLICATION_START_ADDRESS,
        U32_USER_APPLICATION_ALLOCATED_SIZE,
    },
    {0},
    {0},
};
```

- Updates to be performed in user application linker file
  - Add in Memory Space definitions

```
ApplicationFooter(rx) : ORIGIN = 0x005FFF00, LENGTH = 0x00000100
```

  - Add in Section definitions

```
.ApplicationFooter :
{
    KEEP(*(.application_footer_data . application_footer_data.*))
} > ApplicationFooter
```

Reserving RAM Location

The user application and bootloader uses few bytes of RAM to control applications switching between them. Both user application and bootloader must reserve the same location in both applications to avoid conflicts. In the example project provided, 32 bytes of RAM is reserved for switching. This can be done with the following modifications in linker file.

```
ram (rwx) : ORIGIN = 0x20400020, LENGTH = 0x0005FFE0
```

The value of `ram` ORIGIN is changed from default 0x20400000 to 0x20400020, which restricts linker from using the first 32 bytes of RAM.

Fill Unused Locations with 0xFF

Filling unused locations with known data (0xFF) helps tool and bootloader to synchronize during Hash functions. The Hash value generated by tool must match with the value generated by the bootloader. Otherwise, the bootloader assumes the user application is invalid.

To Fill the unused locations with known data:

1. Create a dummy section in the user application.

```
/** Creating a dummy section to fill unused flash with 0xFF.. */
const U8 u8Dummy __attribute__ ((section(".fill"))) = 0xFF;
```

2. Fill this section with known data:

   2.1. Allow linker to relocate initialized data.

```
.relocate :
{
    . = ALIGN(4);
    _srelocate = .;
    *(.ramfunc .ramfunc.*);
    *(.data .data.*);
    . = ALIGN(4);
    _erelocate = .;
} > ram AT > rom
```

   2.2. Fill 0xFF till Application Footer.

```
.Fill_FF :
{
    KEEP(*(.fill))
   FILL(0xFF);
    . = LOADADDR(.ApplicationFooter);
} AT > rom
```

Define Sequence for Bootloader Switching

This is another important step. The user application must define the switching sequence to the bootloader. In the example project provided, this is achieved with the help of the SW0 button press.

When this event is received, the user application calls `bootloader_interface_bootloader_jump_handler` to update the Jump Signature with the `"stay_in_bootloader"` string to remain in the bootloader.

On soft reset, the device starts executing from the bootloader and remains in the bootloader.

4.3.3.5. Modules Description

Bootloader Application Modules

The following table describes functions associated with the bootloader.

Table 4-3. Functions and Descriptions

Function	Description
`Bool bootloader_interface_is_it_required_to_stay_in_bootloader(void)`	This function checks if the user application is requested to stay in the bootloader. • Check if the Jump signature is updated to remain in the bootloader • Clear this instruction to avoid reusing it
`Bool bootloader_interface_is_user_application_valid(void)`	This function verifies the user application. • Get application header data • Check if the application footer is populated or not • Check Integrity of the application • Indicate applications validity
`void bootloader_interface_bootloader_jump_handler(void)`	This function enables calling application to jump to bootloader by updating Jump Signature. • Update Jump Signature
`void bootloader_interface_jump_to_user_application(void)`	This function jumps to the user application after taking the necessary steps in the bootloader. • Release resources before jumping to User Application • Disable interrupts • Get User Application header data • Update vector table • Update stack pointer • Call application reset handler
`U8 flash_interface_erase_user_application(void)`	This function takes care of erasing user application. It does: • Identify Pages and Sectors to Erase • Get No of Pages to Erase • Erase Pages/Erase Sector

Function	Description
`U8 flash_interface_write_to_flash(U32 u32_address, U8* pu8_data_buffer, U16 u16_data_size)`	This function checks whether address is on page boundary or not. Based on that it: • Checks Address is within the user application area • Checks if Address is Page aligned or not • Checks if Data size is matching with Flash Page size • Triggers Write request for the entire page
`U8* aes_interface_decrypt_received_page_data(void* p_data, U16 u16_data_size)`	This function Decrypts input data of size `u16DataSize`. • Enable peripheral clock • Configure the AES in CBC mode • Update Initial Vector • Loop through for all data • Feed in initial set of data • Read Decrypted data
`void icm_interface_calculate_icm(U32 u32_start_address, U32 u32_size, U32* pu32_result)`	This function executes SHA-1 algorithm and updates the result to the output buffer. • Set region descriptor start address • Enable ICM • Check if the region hash is completed

User Application Modules

The following table describes functions associated with the bootloader in the user application.

Table 4-4. Functions and Descriptions

Function	Description
`void jump_to_bootloader_handler(void)`	This function enables calling application to jump to the bootloader. • Get application data from the user application Footer • Get bootloader application Data from the bootloader Footer • Set to remain in bootloader by calling the application jump handler

TFTP Client Modules

The following table describes functions associated with the TFTP client implementation in the bootloader.

Table 4-5. Functions and Descriptions

Function	Description
`void tftp_client_connect_to_server(void)`	This function connects to the server using UDP. It initiates connection with predefined Server IP on Port 69 as per TFTP protocol.
`static void tftp_client_recv_handler(void *arg, struct udp_pcb *pcb, struct pbuf *p, struct ip_addr *addr, u16_t port)`	This is a callback and gets called on receiving a message on TFTP port. • It handles OACK, DATA, and ERROR opcodes in the current implementation
`static void tftp_client_parse_options_ack(struct pbuf *p)`	This function parses options accepted by the Server and sends ack.
`static void tftp_client_process_data(struct pbuf *p)`	This function parses data and sends error or ack. based data processing status
`U8 tftp_client_prepare_options_packet(U8 * pu8_tftp_options, char* pu8_file_name, U32 u32_file_size)`	This function prepares options packet to negotiate with the Server. • Includes “blksize”, “tsize”, and “timeout” into file read request with values requested by the application

4.3.4. Memory Footprint

This section provides memory utilization in bootloader and user application projects.

Table 4-6. Memory Details

Application type	Program memory usage	Data memory usage
Bootloader	38552 bytes	26160 bytes
User Application	19324 bytes	11088 bytes

Note:

- Optimization level (-O1) enabled for both bootloader and user application
- ARM/GNU C Compiler version: 5.3.1, Atmel Studio 7.0, and ASF 3.32.0
- Program Memory Usage bytes are excluding the Fill section, which is filled with 0xFF for Hash functions

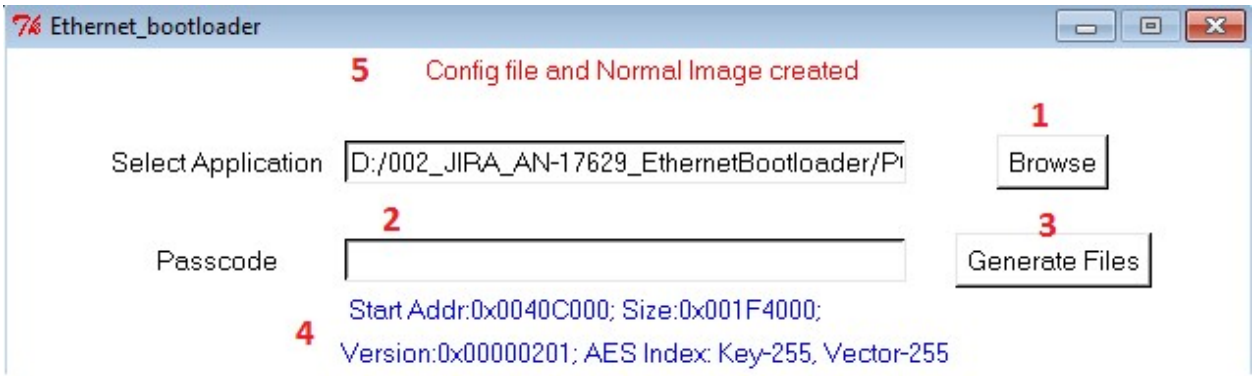
5. Firmware Packager

This helper (Ethernet Bootloader PC tool) program assists to create Configuration and Image binaries to transfer to the bootloader. It is also necessary to have a Server application running on the host. This example project uses a free tftp server (tftpd64.exe) application.

This tool enables the user to:

- Choose user application to create Configuration and Image binaries
- Select AES Key and Initial Vector Indexes for encrypted Image

Figure 5-1. PC Application



The following explains available options and their functionality.

1. **Browse** - Enables the user to select user application bin file.
2. **Passcode** – AES Key and Initial Vectors are predefined in the device and the PC tool. The passcode is a combination of Key Index and Initial Vector Index (0201 – Chooses Key Index as 02 and Initial Vector Index as 01. This example supports four Keys and four Initial Vectors). Leave it blank if no encryption is needed on the selected application file.
3. **Generate Files** – Triggers Configuration and Image binaries creation and Prompts to save the result files. It is required to match these files' name with file names embedded in the bootloader.
   - They are set to `user_app_upgrade_config.bin` and `user_app_upgrade_image.bin` in this example project.
   - The bootloader uses these file names while sending a RRQ request to the TFTP server. Currently these file names are hard-coded in the bootloader.
4. **Configuration Information** – Displays application Start address, size, version, and Selected AES information.
5. **Status Information** – Displays status information while processing user inputs.

5.1. Generate Configuration and Image Files

To generate Configuration and Image binaries, run the Ethernet Bootloader PC tool.

1. Choose the user application file by using the **Browse** option.
2. Choose **Passcode** if encryption is needed.
3. Click on **Generate Files**. Provide location to save the `user_app_upgrade_config.bin` and `user_app_upgrade_image.bin` files.

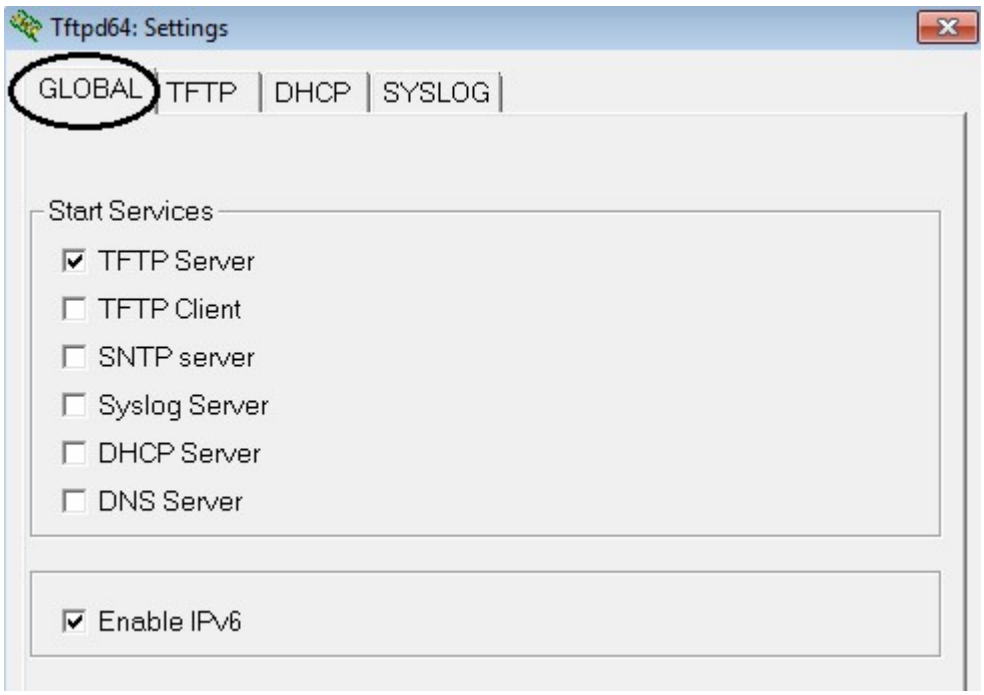
Now, the Configuration and Image binaries are ready for TFTP Server. Copy these binaries into the TFTP Server file location.

5.2. Setup tftpd64 Server Application

This section provides steps to set up tftpd as TFTP Server.

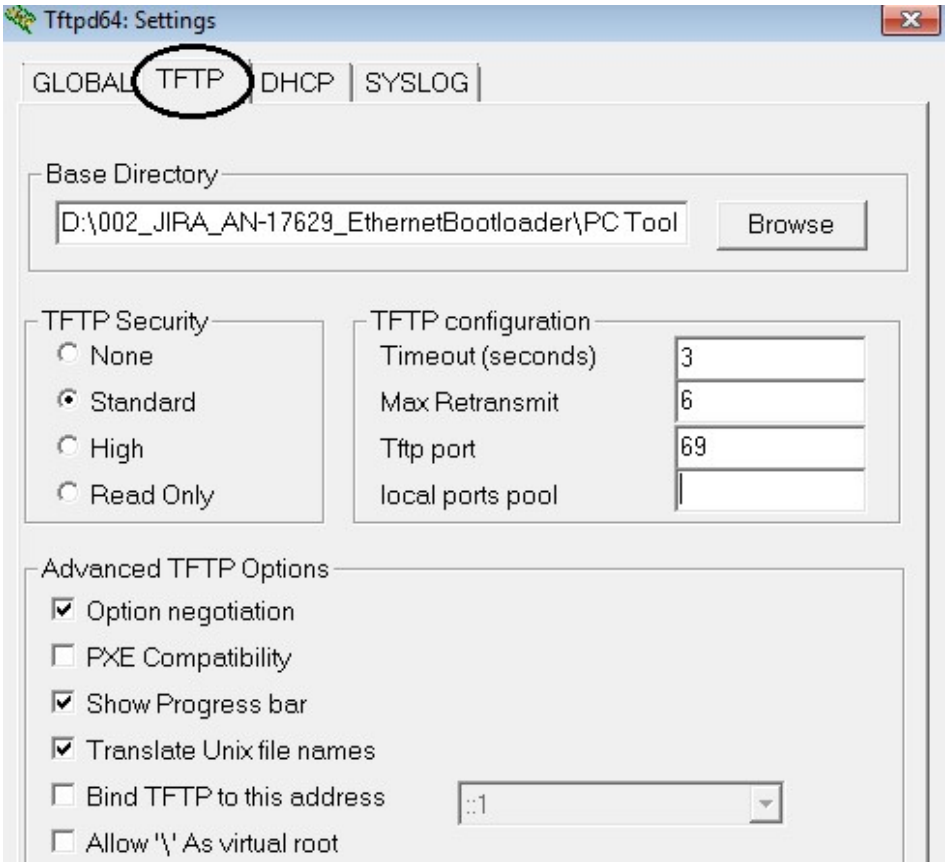
1. Download tftpd32.exe for 32-bit Windows® OS or tftpd64.exe for 64-bit Windows OS.
2. Run tftpd32.exe or tftpd64.exe.
3. Click on **Settings** and select options in the **Global** and **TFTP** tabs.
   - 3.1. **Global** - To start services, click tick box next to the **TFTP Server** and **Enable IPV6** only.

Figure 5-2. TFTPD Global Tab Settings



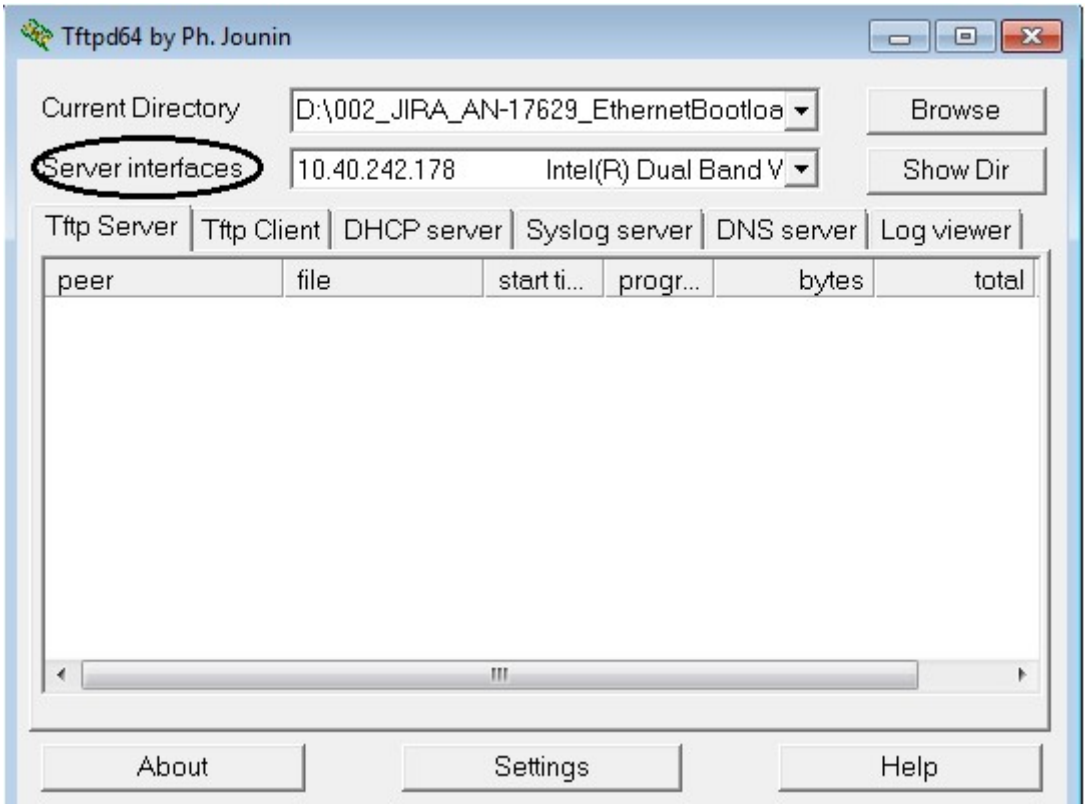
   - 3.2. **TFTP.**
     - 3.2.1. Click on **Browse** to choose **Base Directory**. This should be the directory that contains the `user_app_upgrade_config.bin` and `user_app_upgrade_image.bin` files.
     - 3.2.2. **TFTP Security** - Click the tick box next to **Standard**.
     - 3.2.3. **TFTP configuration** - Set **Timeout (seconds)** as 3, **Max Retransmit** as 6, and **Tftp port** as 69.
     - 3.2.4. **Advanced TFTP Options** - Click the tick box next to **Options negotiation**, **Show Progress bar**, and **Translate Unix file names**.

Figure 5-3. TFTPD TFTP Tab Settings



4. Click **OK**.
5. Close exe and restart once again.
6. Select **Server interfaces** to network in which the TFTP client is connected. The following figure illustrates the same.

Figure 5-4. TFTPD Server Interfaces Settings



6. Running Example Bootloader and User Application

The following points provide steps to experience the example bootloader and the user application provided along with this application note.

1. Download tftpd32 or tftpd64 exe from web.
2. Set up the tftpd server application as explained in Setup tftpd64 Server Application.
3. Open SAME70_TFTP_Bootloader.atsln by using Studio.
   - 3.1. Update TFTP Server IP in bootloader. (Update `TFTP_SERVER_IP_ADDRESS0`, `TFTP_SERVER_IP_ADDRESS1`, `TFTP_SERVER_IP_ADDRESS2`, and `TFTP_SERVER_IP_ADDRESS3` macros.)
   - 3.2. Compile the project.
4. Open SAME70_Getting_Started_ASF_Example.atsln by using the Studio and Compile project.
5. By using the Ethernet Bootloader PC tool, generate `user_app_upgrade_config.bin` and `user_app_upgrade_image.bin` files as explained in Generate Configuration and Image Files.
6. Copy configuration and image binaries to the TFTP server file directory.
7. Download the binary by using the EDBG port on the XPLD board or the programmer available.
8. Connect the Ethernet Cable to establish connection between the server and client and power the Cycle XPLD board. This should start.
   - 8.1. Acquiring an IP address from the DHCP server.
   - 8.2. On getting the IP address, starts the TFTP client interactions with the Server
   - 8.3. First, reads configuration file and then image file.
9. After completing the firmware upgrade, the control starts executing the user application.

At the end of this, you should see LED0 toggling on the XPLD board. To be able to enter into bootloader, press the SW0 button on the XPLD board.

Note: Both the bootloader and the user application projects print information on the console window. One can use Hyper Terminal or equivalent to view these messages on the EDBG port.

7. References

1. Federal Information Processing Standards 180-2 - http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.
2. The TFTP Protocol - https://tools.ietf.org/html/rfc1350.

8. Revision History

Doc. Rev.	Date	Comments
42782A	09/2016	Initial document release

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**