# PROGRAMMING MODULE USING UART OR SPI

| Version | Date | Remarks |
|---|---|---|
| 1.0 | May 2011 | ▶ Initial Release |
| 1.1 | April 2012 | ▶ Updated with Memory Map Info<br>▶ Added additional info on the HI format and Additional Info field<br>▶ Added info on MAC Address |

# Table of Contents

# Figures

# Tables

# 1 Introduction

The GS1011M or GS1500M module uses the GainSpan GS1011 system-on-chip device. On the GS1011, there are two CPU: WLAN CPU and APP CPU. The WLAN is the main CPU that first runs after power up and boots from its internal boot ROM. The code in the boot ROM first checks if a firmware download is requested. This is achieved by reading the state of GPIO27 (pin 36 of the module). At power-on or reset, if the GPIO27 is high, then the module is in program mode. Once in program mode, either UART0 or Slave SPI (SSPI) interface can be used to program the module flash.

The UART0 interface would run with the following setting:

Baud rate: 115200 kbps

Data: 8 bits

Parity: None

Stop: 1 bit

Flow control: None

The Slave (SSPI) would be running at higher speeds up to 1.5MHz.

## 1.1 Power On

The WLAN CPU always boots from its internal boot ROM. The code in boot ROM performs the following operations:

1. After reset or power-on, the boot code first checks if a firmware download is requested. This is achieved by reading the state of GPIO27. If GPIO27 is high, the boot code proceeds to step 3. Otherwise, it goes to step 2.
2. If GPIO27 is low, the boot code checks the validity of the firmware present in flash memory, by reading the firmware signature in the information section of the memory bank. If a valid signature is detected, then the program proceeds to step 4. Otherwise it goes to step 3.
3. If GPIO27 is high, or no valid signature is found, the boot code waits for code to be received on the UART or SSPI interfaces, according to the Firmware Update procedure. When the firmware update procedure is complete, the boot code proceeds to step 4.
4. Boot code jumps to the start of WLAN Firmware (WFW) in WLAN flash memory.

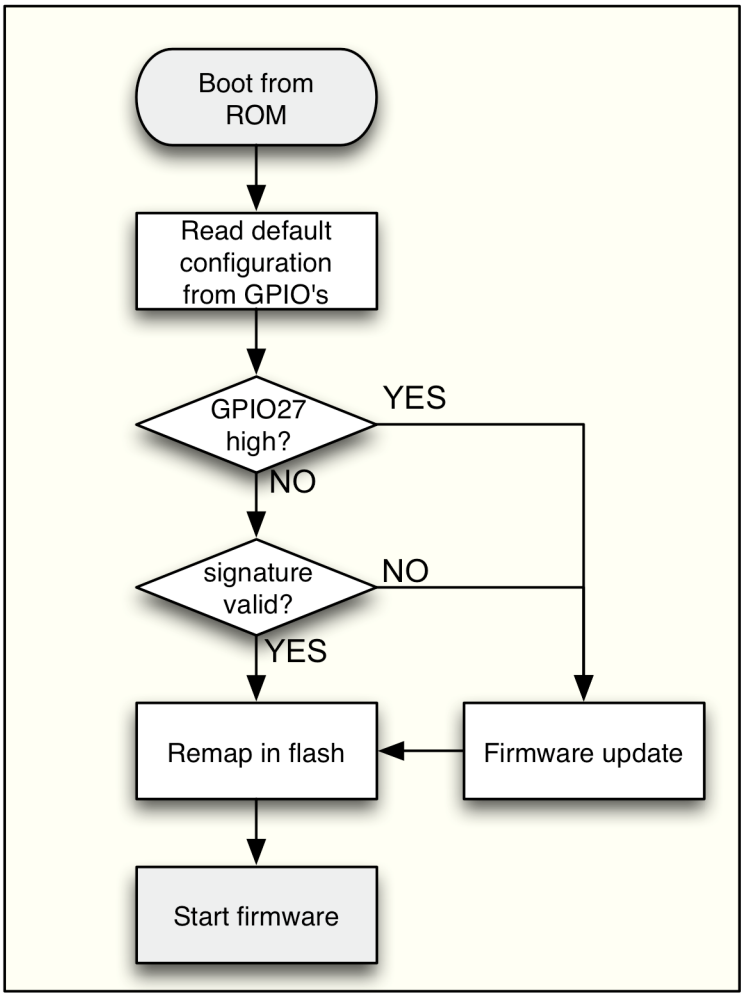



**Figure 1: Boot ROM Operation Flow**

Programing the device over the host interface (UART0 or SSPI) uses the following protocol format for transferring data.

# 2 Host Interface

The module provides a Host Interface, which allows the on-Chip APP CPU or an external CPU to do programming operations through a message based protocol on either the Slave SPI or UART0 physical interface..

Messages are organized into ***service classes***, according to the general function they support. Each class has a slightly different message format and a different usage model. The classes are:

- **Kernel message service**: messages controlling WFW operation and configuration
- **Data service** (and **Data Confirmation service**): messages whose contents are intended to be sent or received over the wireless link (not described here)
- **Download service**: messages used to write new firmware (or other data) to the WLAN and APP Flash memory banks
- **Debug service, Trace service**: (not described here)
- **ROM Debug service**: messages that allow direct read and write to/from memory locations and memory-mapped registers (not described here).

Messages can be sent and received over differing physical-layer transport mechanisms – Mailbox, UART0, Slave SPI, and WLAN – although not all service classes can be used with all PHY layers (see Table 1). Messages consist of a PHY-independent Host Interface header and data section, with additional data specialized for the relevant PHY layer. The focus here is going to be the UART and Slave SPI interface for programming of the module.

An example of how an abstract message is reduced by the protocol stack to a byte sequence is depicted in Figure 2. An internal module (the Scan module) needs to tell the host processor that it has found four stations in response to a request to scan for WLAN access points. This message is of the service class ***kernel messages***. The sending module, the Scan module, is the ***source*** of the information. Since the message is destined for an external processor, its formal ***destination*** is the WFW Host Interface module (which is responsible for forwarding the message over the appropriate physical layer). The kernel message, consisting of a message identifier, destination and source modules, a length field, and parameter values appropriate to the particular type of message, is formed according a class-specific format to create an ***HI Parameters*** field. This parameter field is passed to the Host Interface, which adds a physical-layer-independent header describing the class and length of the HI Parameters field. (Some classes also make use of the Additional Info field of the header.) The resulting HI frame then receives any necessary additions specific to a given physical layer; for the UART PHY, a start-of-frame (SOF) delimiter and header checksum packet are used.

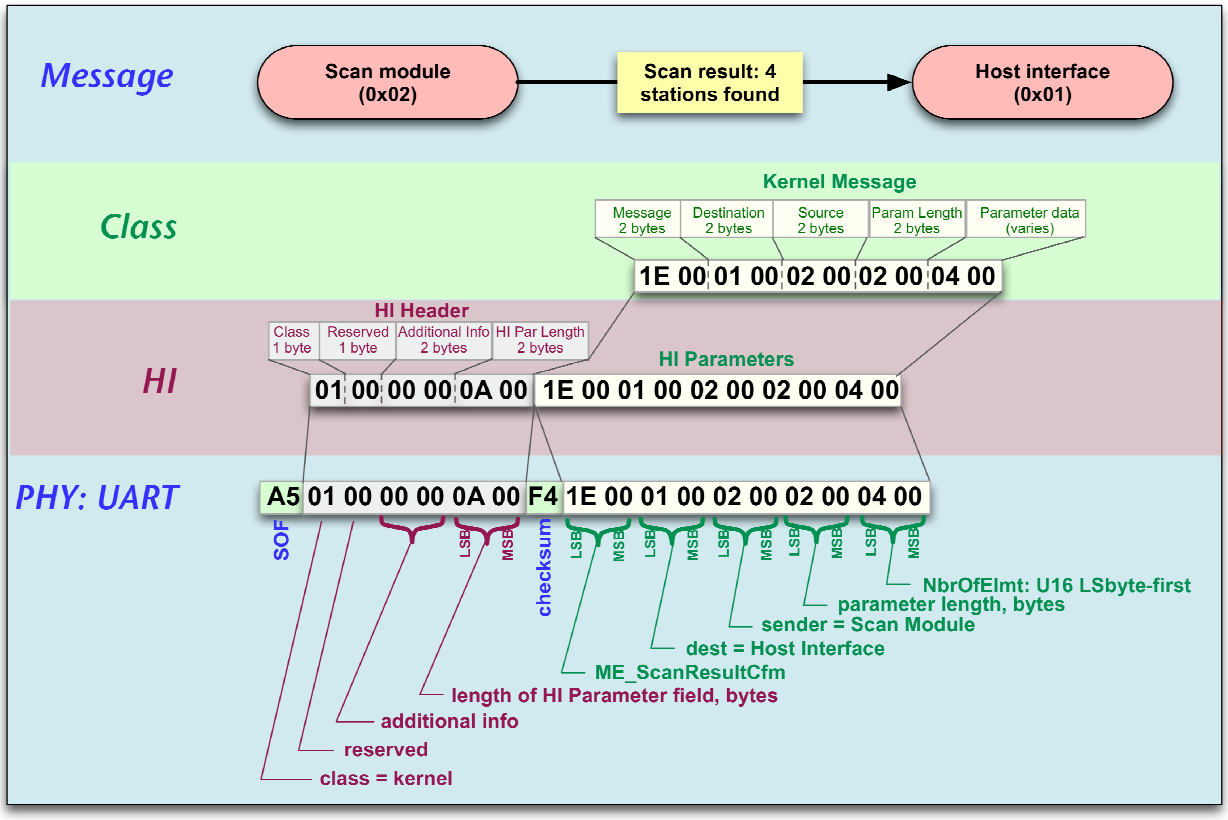



**Figure 2: Host Interface Protocol Stack Example**

The HI interface and specializations to various PHY layers are described in detail in the remainder of this chapter.

## 2.1 HI Overview

The Host Interface (HI) is the software layer that enables access to the WLAN services that are provided to an external CPU by the WLAN CPU.

These services are offered over the various external interfaces of the WLAN CPU: the Mailbox interface (or more specifically the Mailbox Frame Exchange mechanism), the SPI slave interface, the UART interfaces and the WLAN interface (for the firmware update procedure).

The Mailbox Frame Exchange was specifically designed to carry HI frames unmodified; messages must be encapsulated in larger frames to be sent over the other interfaces. The required encapsulation will be defined in this chapter.

The Mailbox Frame Exchange is the default interface for the HI frames. The use of other PHY layers is approached differently depending on whether the frames in question are sent by the Host, or sent by the WLAN CPU to the Host.

The WLAN CPU is always waiting for incoming frames on all the interfaces that it has control of, so an incoming HI frame will always be handled, regardless of the interface used.

Figure 3 provides a classic OSI-style view of the protocol stack.

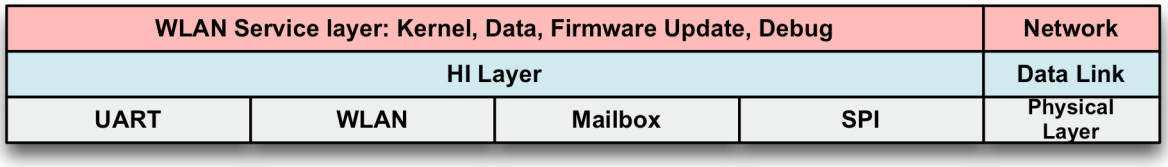



**Figure 3: OSI-like Presentation of the WLAN Service Stack**

## 2.2 HI Frame Format

All messages carried over the Host Interface have a common format. They are composed of an HI header, and parameters depending on the header. This format is defined in Figure 4. The parameter field maximum length is chosen to allow Ethernet frames (source, destination, and payload) to be carried inside.

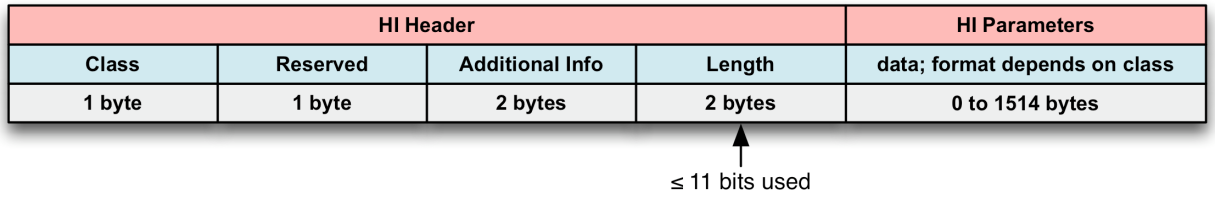



**Figure 4: HI Frame Format.**

The format of the HI Parameters field is determined by the service class. The service class of each frame is signaled by the value of the first field; available service class identifiers are:

- 0x00 : Data (and Data Confirmation) service.
- 0x01 : Kernel message service.
- 0x02 : Download service.
- 0x03 : Debug service.
- 0x04 : reserved
- 0x05 : reserved
- 0x06 : Trace service.
- 0x07 : ROM Debug service.

The Additional Info field is a general-purpose field whose contents depend on the service class of the message.

The length field is 2 bytes long even though only 11 bits will be used, since the length cannot exceed 1514. The length value does not include the 6 bytes of the HI header. Depending on the PHY used to carry the HI frame, it is possible that only the 11 significant bits will be transmitted.

All the service classes are not intended to be carried over all the interfaces. The service classes supported by each interface are listed in Table 1.

**Table 1: Supported Service Classes for UART and Slave SPI Interfaces**

| PHY | Kernel | Data | Debug | Download | Trace | ROM Debug |
|---|---|---|---|---|---|---|
| **UART** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Slave SPI** | Yes | Yes | Yes | Yes | Yes | Yes |

### 2.2.1 Transmission and reception

The HI frame exchange can be built on top of various communication interfaces: Mailbox frame exchange, UART, SPI or the 802.11 wireless link.

When a WFW task needs to send a frame, the sending task adds a descriptor in the HI transmission FIFO that is used to store pending HI frames. When a new transmission can be scheduled on the selected transmission interface, an HI frame is retrieved from the transmit FIFO and sent on that interface.

In reception, the various enabled interfaces will push their received frames to a receive FIFO. The handler then regularly polls this FIFO to see if there are HI frames pending, retrieves those frames if necessary, and handles them.

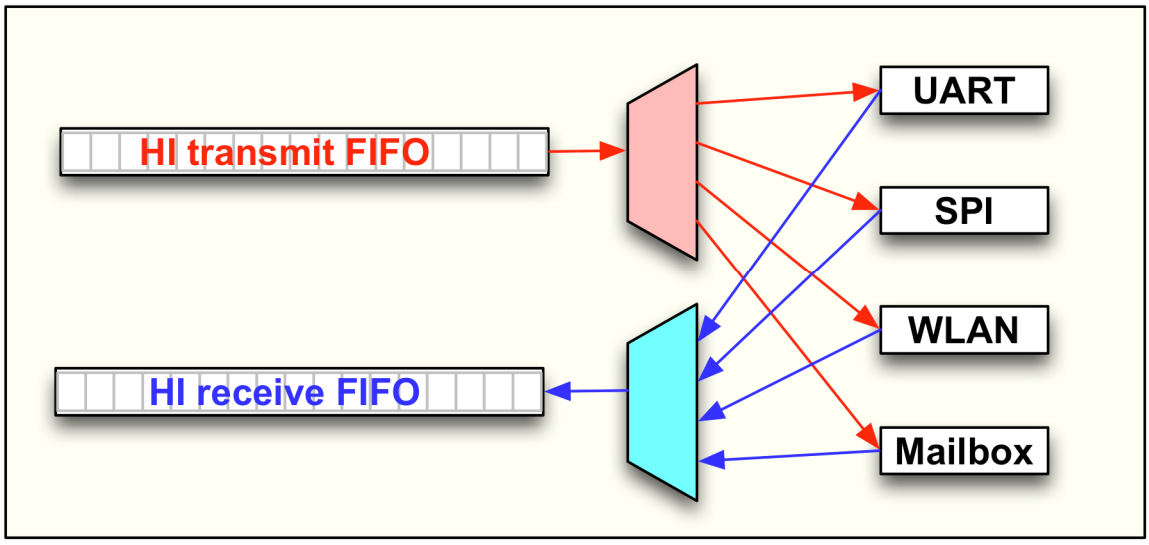



**Figure 5: HI FIFOs**

## 2.3 HI frame Mapping over Physical Interfaces

The format of the HI frame depends on the transport layer used. Below describes the format for the UART and Slave SPI interface.

### 2.3.1 HI Frame over UART

HI frames carried over the UART interface are composed, in addition to the HI header and parameters, of a start delimiter and an HI HEADER checksum (**Error! Reference source not found.**). Numerical values are sent least-significant-byte (LSByte) first. Delimited byte fields, such as MAC addresses, SSID's, and BSSID's, are sent in the conventional order in which they would be read (that is, with the leftmost byte or character sent first).

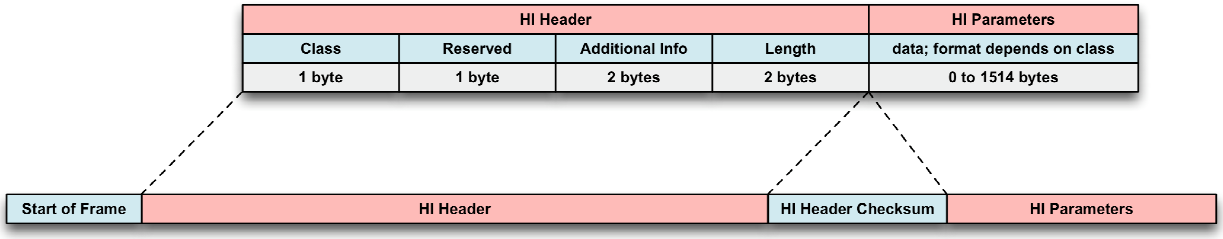



**Figure 6: HI Frame Format over UART**

The Start-of-frame delimiter is the single-byte value 0xA5 (**Error! Reference source not found.**), used to ensure synchronization at the frame level. The driver starts the reception process when it recognizes the delimiter.

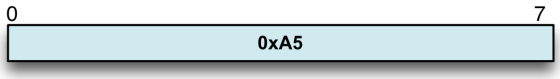



**Figure 7: Start-of-Frame Delimiter**

The length of the delimiter has been reduced to 1 byte to avoid alignment problems when waiting for the start element. However, no provisions are made to ensure that the subsequent data stream does not contain a byte with value 0xA5, so it is possible for the driver to mistake a data byte for a delimiter. Therefore, a header checksum has been added to ensure correct synchronization. The checksum definition is shown in **Error! Reference source not found.**. A single checksum byte is used, computed as the 1's complement of the 8-bit long (modulo-256) sum of all the bytes of the HI HEADER (not including the Start delimiter). Note that each byte is independently added to the sum, as an integer between 0 and 255, without regard for its significance within its own data field.


**Figure 8: HI Header Checksum**

### 2.3.2 HI Frame over Slave SPI

HI frames carried over the Slave SPI are composed, in addition to the HI header and parameters, of a start delimiter and an HI HEADER checksum:

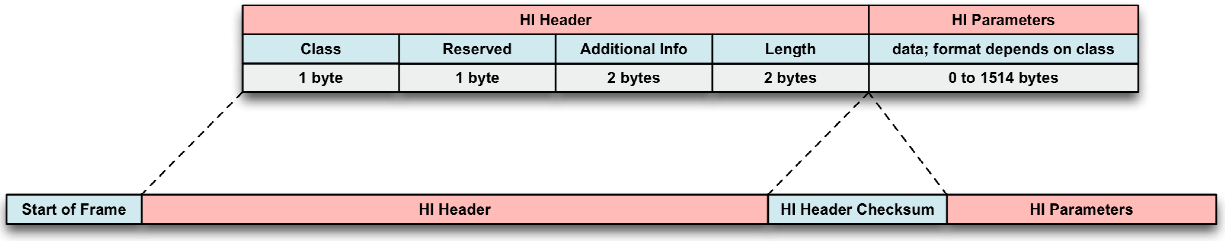



**Figure 9: HI Frame Format over SPI**

The Start-of-frame delimiter is the single-byte value 0xA5 (**Error! Reference source not found.**), used to ensure synchronization at the frame level. The driver starts the reception process when it recognizes the delimiter.

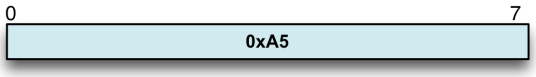



**Figure 10: Start-of-Frame Delimiter**

The length of the delimiter has been reduced to 1 byte to avoid alignment problems when waiting for the start element. However, no provisions are made to ensure that the subsequent data stream does not contain a byte with value 0xA5, so it is possible for the driver to mistake a data byte for a delimiter. Therefore, a header checksum has been added to ensure correct synchronization. The checksum definition is shown in **Error! Reference source not found.**. A single checksum byte is used, computed as the 1's complement of the 8-bit long (modulo-256) sum of all the bytes of the HI HEADER (not including the Start delimiter). Note that each byte is independently added to the sum, as an integer between 0 and 255, without regard for its significance within its own data field.


**Figure 11: HI Header Checksum**

## 2.4 Download Protocol

### 2.4.1 Message Flow

During a firmware update, the new firmware code is first stored in the shared SRAM and verified using a checksum. Once validation has occurred, the relevant flash memory block is erased and the code is copied into it.

Figure 12 depicts the sequence of messages exchanged between the host and the WLAN CPU (the ***Download/Copy*** sequence). If both the WLAN and APP CPU firmware are to be updated, or if the RAM size is not sufficient to perform the update in one sequence, the sequence below is repeated for the required number of times, as described below. The on-chip flash memory is organized in 2kB sectors, so the firmware fragment downloaded in one Download/Copy sequence must be aligned on a 2kB boundary, and has a length that is a multiple of 2kB, except for the last fragment.

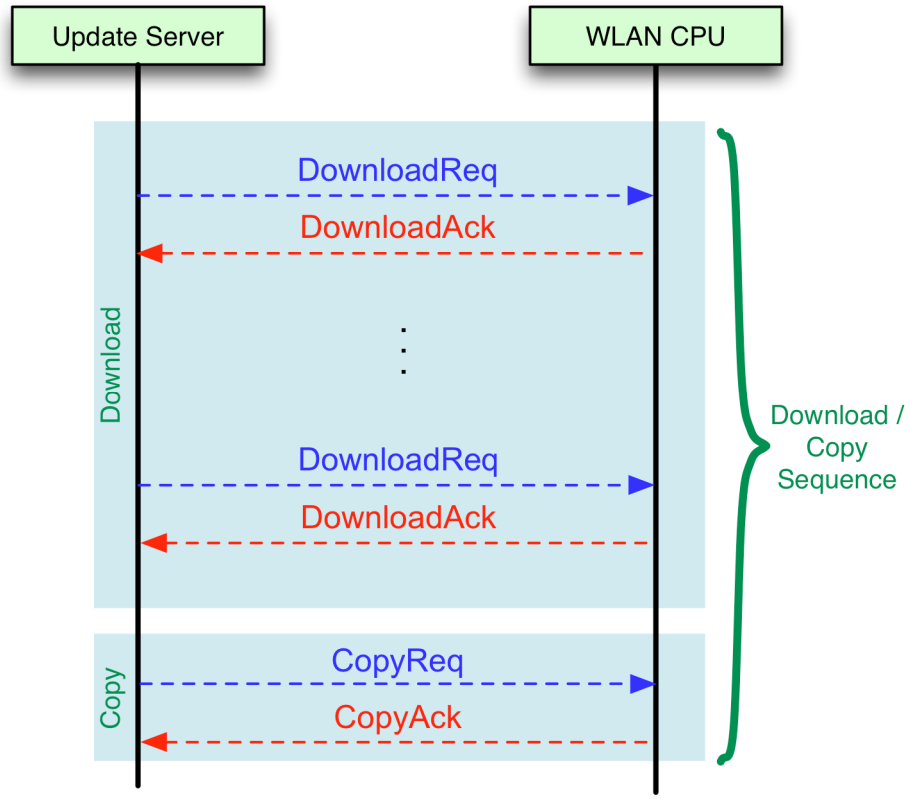



**Figure 12: Firmware Update Message Flow**

Note: The DownloadAck frame indicates the proper reception of the Download frame, not the proper execution of the associated command. This mechanism is used for both flow control and reliability purposes.

## 2.4.2 Message Format

The firmware update message is contained in an HI frame that has the DOWNLOAD class (Figure 13).

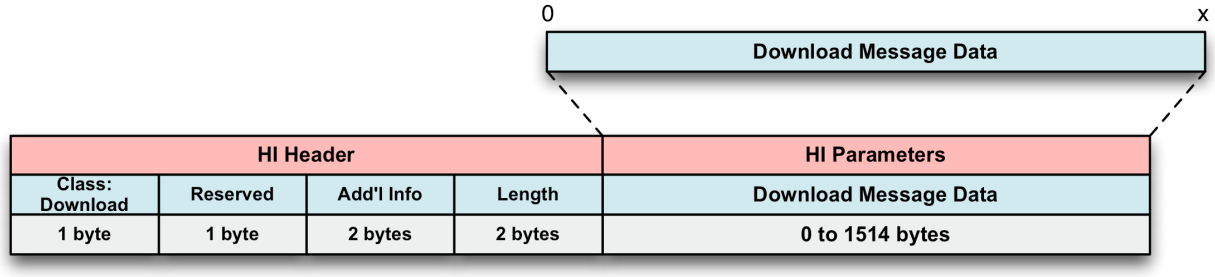



**Figure 13: Firmware Update Message Mapping on HI Frame**

Note: For Download class, Additional Info field is not used and can be 0000

### 2.4.2.1 DownloadReq

The Download request is composed of six fields (Figure 14):

- *Opcode*, 7 bits: 0x01. The least-significant bit is bit 0 in the diagram. Bit 7 is reserved.
- *SeqNbr*, 1 byte: Sequence number of the current request
- *FileLength,* 4 bytes: Total length of the downloaded firmware.
- *BundleLength*, 2 bytes: Length (in bytes) of the *Data* field
- *RamAddr*, 4 bytes: Address where the current code bundle must be stored in shared SRAM
- *Data*, <*BundleLength*> bytes: Data to be written to SRAM

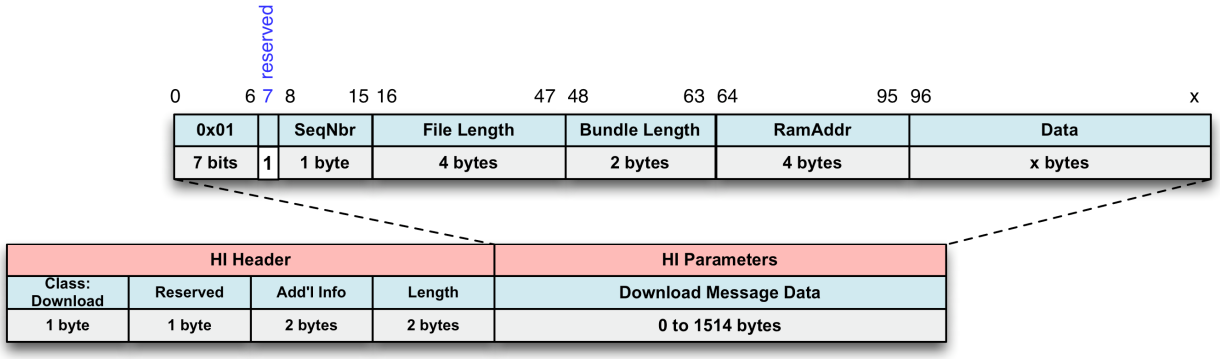



**Figure 14: Download Request Format**

Note: Additional Info field is not used and can be 0000

### 2.4.2.2 CopyReq

The Copy request is composed of eight fields (Figure 15).

- *Opcode*, 7 bits: 0x02.
- *More*, 1 bit: Bit 7 in the diagram, indicating if another download sequence is still pending

- *SeqNbr*, 1 byte: Sequence number of the current request
- *FlashAddr*, 4 bytes: Destination address (in flash memory) of the buffer to copy
- *Length*, 4 bytes: Length (in bytes) of the buffer to copy
- *RamAddr*, 4 bytes: Source address (in SRAM) of the buffer to copy
- *Checksum*, 4 bytes: Checksum of the downloaded data
- *SwapFlashAddr,* 4 bytes: Address of the flash memory that is to be used as Swap Flash

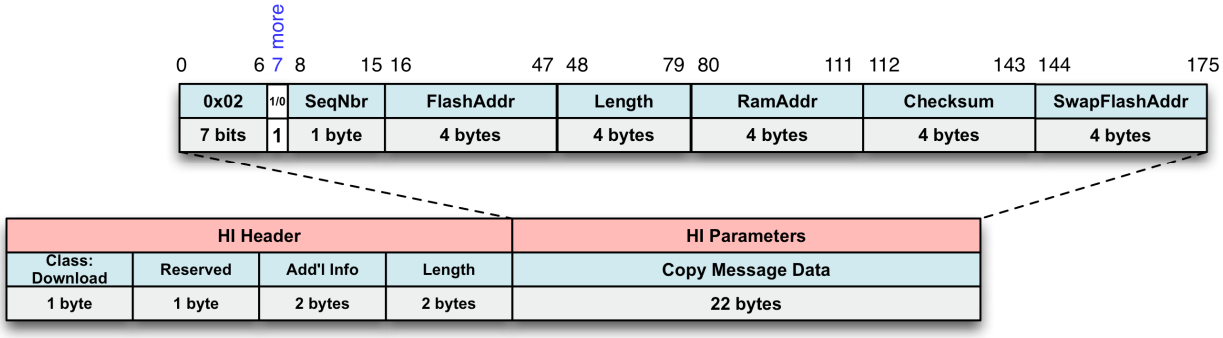



**Figure 15 Copy Request Format**

Note: Additional Info field is not used and can be 0000

When receiving this request, the WLAN CPU performs checksum verification on the downloaded data using the *Checksum* parameter (see 2.4.3 for details about the checksum calculation), erases the flash sectors affected by the download, and copies the data from SRAM to Flash according to the *RamAddr*, *FlashAddr* and *Length* parameters.

If code checking is ok and no download sequence is still pending (indicated by the *More* bit), the execution of the new firmware is started. If an invalid checksum is detected, the download error procedure is performed (see 2.4.4). If the *More* bit is set, then the WLAN CPU waits for subsequent *Download* requests.

The *SwapFlashAddr* parameter is used only in the case where the WLAN firmware is being updated, and the shared RAM size is 64 KB. In that case the host must indicate, using *SwapFlashAddr*, in which APP Flash block the WLAN firmware must be copied temporarily before being copied into the WLAN Flash.

#### 2.4.2.3 ErrorInd

The Error indication is composed of three fields (Figure 16):

- *Opcode*, 7 bits: 0x04
- *SeqNbr*, 1 byte: Sequence number of the current indication
- *ErrorCode*, 2 bytes: Failure code:
    - 0x01: ModuleBusy. The Firmware Update module was not ready to receive a command. The Host can retry the command.
    - 0x02: ParamError. The CopyReq contains inconsistent parameters.

- 0x11: TimeOut. Activity timeout expired.
- 0x12: ChecksumError. The computed checksum is different from the one passed in the CopyReq.
- 0x13: FlashError. The flash memory erase or write has failed. In this case, the MSB of ErrorCode indicates which block failed: 0x01 for WLAN, 0x02 for APP0, 0x03 for APP1).
- 0x14: SeqNumError. A bad sequence number has been detected.
- 0x15: RAMOverlap. The address passed in the DownloadReq is bigger than the Shared RAM size.

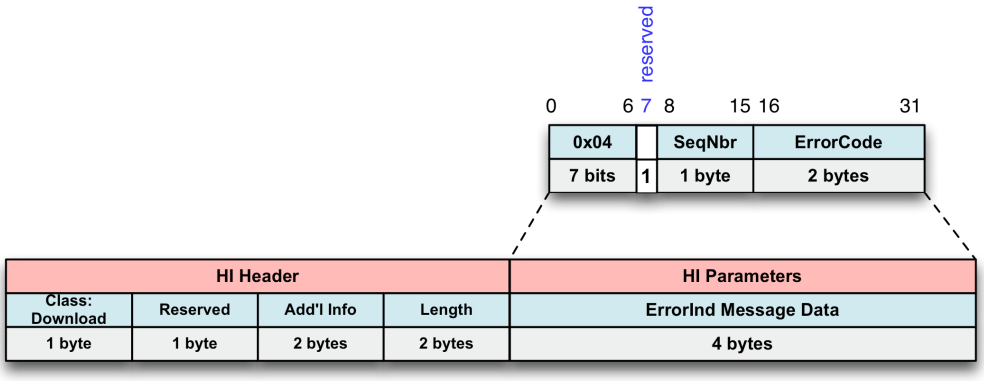



**Figure 16: Error Indication Format**

Note: Additional Info field is not used and can be 0000

### 2.4.2.4 Ack

The Ack message is composed of two fields (Figure 17):

- *Opcode*, 7 bits: 0x05
- *SeqNbr*, 1 byte: Sequence number of the request being acknowledged.

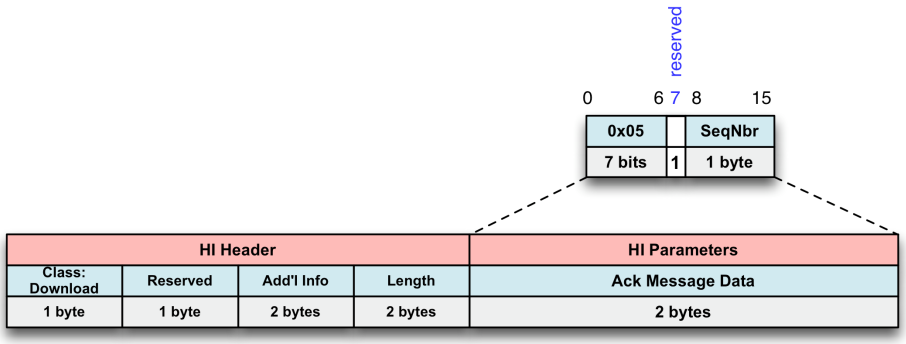



**Figure 17 Ack Format**

Note: Additional Info field is not used and can be 0000

### 2.4.3 Checksum Verification

The firmware validity is checked using a 32-bit checksum. The checksum computation is performed by adding the bytes of the firmware in a 32-bit word, and taking the 1's complement of the result. The expected checksum is sent to the WLAN CPU in the *Copy* request. In the case the firmware update is performed using several *Download/Copy* sequences, a checksum must be calculated for the firmware fragment downloaded during each sequence.

### 2.4.4 Download Error Management

If a failure occurs during the download (such as a timeout due to disconnection, a firmware checksum failure, a problem with the flash memory, and so on), an ErrorInd message is transmitted to the host, indicating the reason of the failure. Depending on the time where the failure occurs, different actions can then be performed.

- If the APP Flash is still valid then the APP CPU is restarted and warned about the failure using the information registers.
- If the APP Flash is invalid (that is, the error occurred during APP Flash update), then the WLAN CPU remains in Firmware Update mode and waits for subsequent update requests from the Host.

### 2.4.5 Firmware Signature

Before erasing a flash block (WLAN or APP0/1), the WLAN CPU invalidates the Flash by erasing the Firmware signature located in the information block of the block. Once the block has been successfully erased and programmed with the new firmware, the WLAN CPU rewrites the Firmware signature. The use of the signature provides a mechanism to ensure that the processor does not boot from invalid firmware.

### 2.4.6 WLAN Software operation

Figure 18 describes the sequence of WFW operations during a firmware update procedure. The sequence applies to updates over any of the interfaces (wireless or serial).

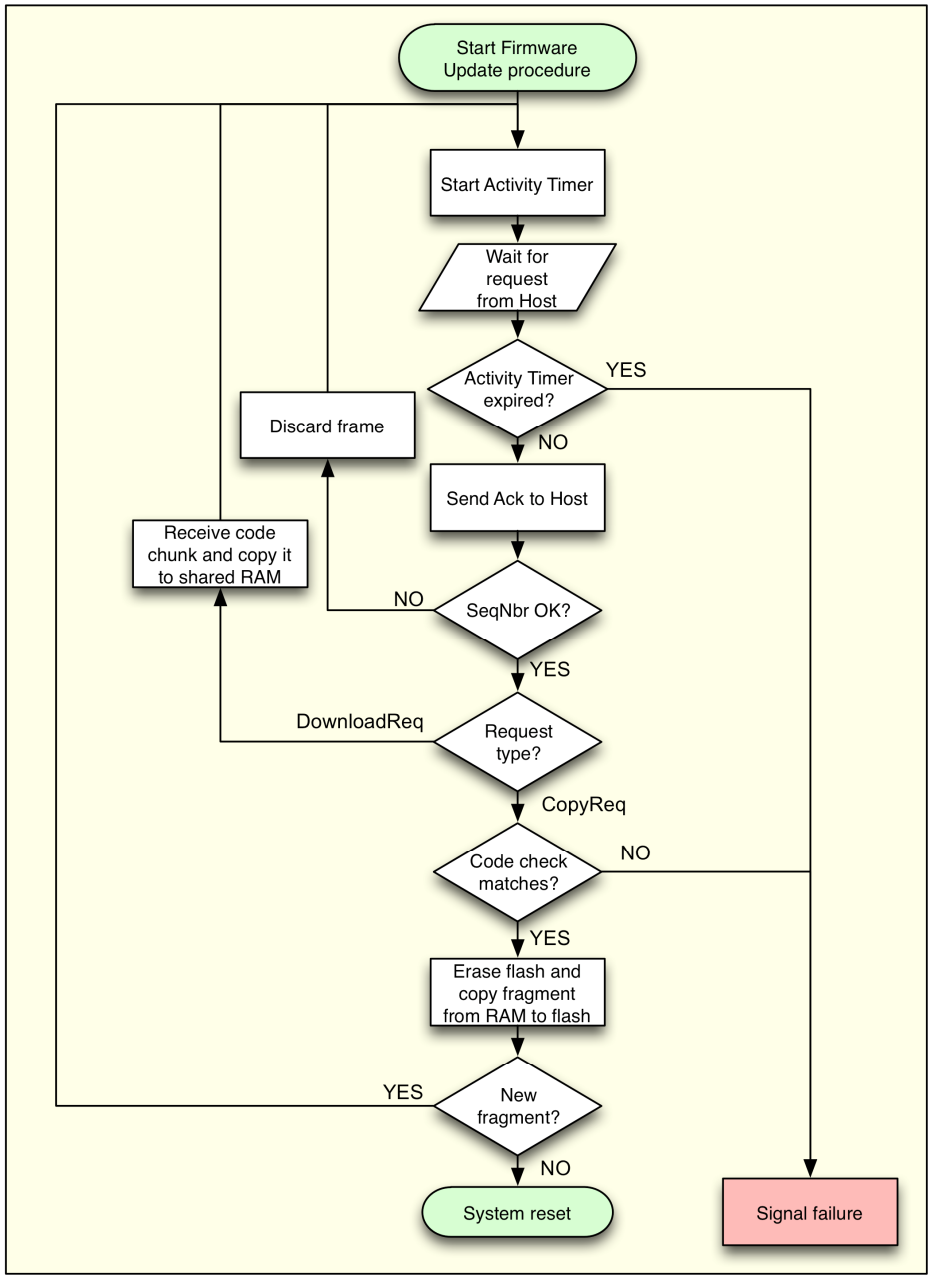



**Figure 18: Firmware Update Procedure**

During a firmware update the following operations are performed:

1. Request waiting
   1.1. The activity timer is started by the WLAN CPU
   1.2. If the activity timer expires, then the WLAN software proceeds to 3
2. Request handling

2.1. Upon reception an Acknowledgement is sent to the host.

2.2. The sequence number of the request is then checked. If the frame has been already received, it is discarded and the WLAN software goes back to 1.

2.3. The request is then executed:

- If this is a DownloadReq, the *Data* field is copied into the shared SRAM at the address indicated in the request. The WLAN software then goes back to 1 to wait for the next request.
- If this is a CopyReq, the downloaded fragment validity is checked. If not valid the WLAN CPU proceeds to 3. If valid the required flash memory is erased, as per the parameters in the request, and the fragment is copied from the SRAM to flash memory. If the *More* bit in the request indicates that a new *Download/Copy* sequence will start then the software goes back to 1. Otherwise it proceeds.
- Firmware start-up. If the CopyReq indicates that it was the last *Download/Copy* sequence, then the new firmware is started by resetting the system.

3. Download error management. The behaviour in case of error depends on the medium for the download (wired or wireless).

## 2.5 Memory Map : Flash Memory and Control Register

The gs_flash program writes to the flash controller first followed by writing info the respective flash blocks.

**Table 2: Module Internal Flash Memory Map**

| | | |
|---|---|---|
| 0x03000000 | 0x03ffffff | WLAN Flash – 128K |
| 0x04000000 | 0x04ffffff | Shared RAM – 128K |
| 0x06000000 | 0x06ffffff | APP Flash 0 – 128K |
| 0x08000000 | 0x08ffffff | APP Flash 1 – 128K |
| 0x05000D00 | 0x05000Dff | WLAN Flash Controller |
| 0x07000300 | 0x070003ff | APP Flash 0 Controller |
| 0x07000400 | 0x070004ff | APP Flash 1 Controller |

## 2.6 MAC Address Overview (in Factory Default Area)

The modules contain a default MAC ID that corresponds to the label on the Module. The MAC Address is the first field in the Factory Default Area and is stored in the following format:

| | | |
|---|---|---|
| Checksum (1 byte) | Length (1 byte) | Mac Address (6 bytes) in HEX |

Checksum : Simple byte wise xor of both length and MAC address

Length: The length in bytes of MAC address and length (here it is 7)

MAC Address: MAC ID (6 byte hex value) that is on the module label

For details see the Serial-to-Wi-Fi Adapter Guide Document.

Note: The MAC ID is the first address in the factory default area.

The MAC ID is located at address 0x0801E800 (Factory Default Start Address) when using firmware version 2.3.x or 3.3.x

# 3 GainSpan Programming Utilities:

GainSpan has developed a programming utility, called gs_flashprogram, that provides both a GUI interface and a DOS command line interface for programming the module. The utility is configurable to use either UART or SPI to program the module.

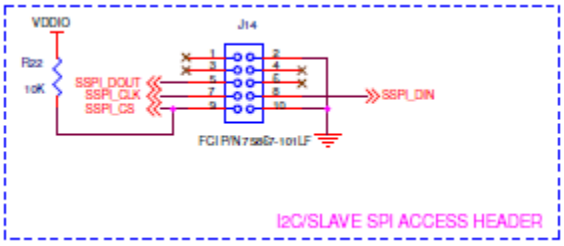
The GainSpan module comes with a Serial to Wi-Fi version pre-programmed into the module. If user is programming a version of GainSpan's Serial to Wi-Fi version of the firmware as is, then user may first check the version already programmed in the module by using the AT+VER=? Command. If the version matches the latest version released by GainSpan, then programming is not needed. If user is programming their own version of the firmware that was developed using GainSpan's SDK, then customer will have to erase and re-flash the appropriate firmwar version into the module embedded flash.

Install the latest gs_flashprogram provided by GainSpan and follow the readme or user manual that comes with the gs_flashprogram utility. The sections below describe the various steps to flash the firmware. If using SPI mode, then see section for additional drivers and software items needed. If using UART0, then user may skip section 2.1.

## 3.1 Pre-requisite for SPI based programming.

This section describes the HW needed and the process for programming using SSPI interface.

A Total Phase Aardvark I2C/SPI Host Adapter (Part Number TP240141), which can be purchased from http://www.totalphase.com/products/aardvark_i2cspi, is recommended to connect to the SSPI signals on the system board. It is also recommended that system design bring out the SSPI pins to either test points or 10-pin header as shown in the circuit below. See the SDK board schematic for more details.




The following steps need to be followed to copy the necessary SW to enable SPI based programming of the module. This is assuming that the customer has already installed the gs_flashprogram installation package. The installation package includes some additional pieces of software such as the WildServer that provides an abstraction layer for SPI and converts the data to the SPI HI format.

1. Please install Total Phase Aardvark driver. You may download the latest software release from Total Phase at: http://www.totalphase.com/products/aardvark_i2cspi/

2. To install the Aardvark driver, first plug in the Aardvark then point the installer to the ***.\GSFLASHPROGRAM\Total Phase_2008*** location. Click "**Continue Anyway**" if you see Figure 1 while installing Aardvark driver. For details see the "Detailed Aardvark Installation" doc.


**Figure 19 Aardvark Driver Installation**

4). Run **WiLDServer.bat** from ***.\GSFLASHPROGRAM\WiLDServer_2_0_15*** folder. Select the connection as SPI. Refer to the figures below for **Wildserver** link with SPI. If **Wildserver** is successfully launched then proceed to next step, else close **Wildserver** recheck Aardvark connection and restart **Wildserver** again.

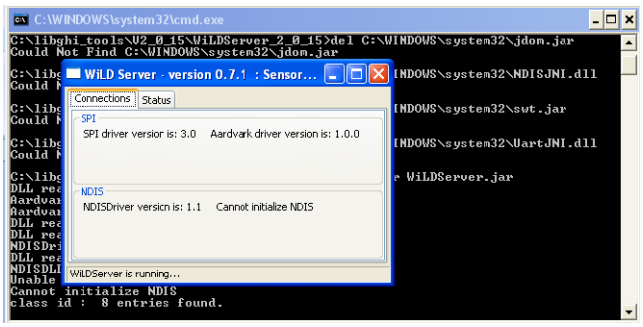



**Figure 20 Wildserver SPI Success**

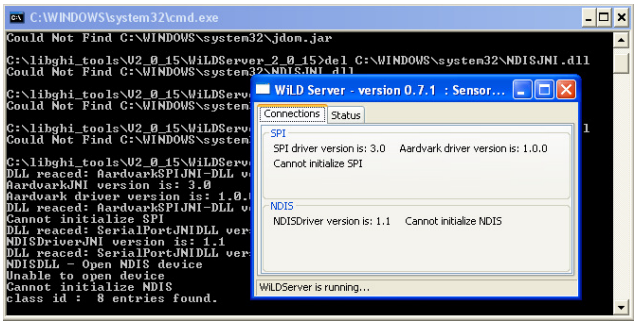



**Figure 21 Wildserver SPI Fail**

# 3.2 Erasing Flash and loading WLAN and APP Firmware

## 3.2.1 Steps for Erasing and loading WLAN Firmware

Note: this method should not be used if using Serial to Wi-Fi or custom firmware and if customer is planning to use the MAC Address already pre-programmed on the module. Instead use method as described in Section 3.3.

Customer may use this process, if they first read the MAC address from the module and save it, then erase the flash and write the saved MAC address back to the factory default area. The MAC ID format and location depends on the firmware version being used as described in Section 2.6.

1) Copy the WLAN, APP0 and APP1 firmware that is targeted to be programmed to .\GSFLASHPROGRAM working directory.

3) Open a dos window and go to .\GSFLASHPROGRAM

4). Type the command ***gs_flashprogram –v –e0 –e1 –ew*** as shown in Figure 22


**Figure 22 Erasing firmwares**

5). This erases the entire flash including the factory default MAC Address

6). To load WLAN type the command ***gs_flashprogram -v –w<WLANfirmware_file_name>.bin*** as shown in Figure 23.

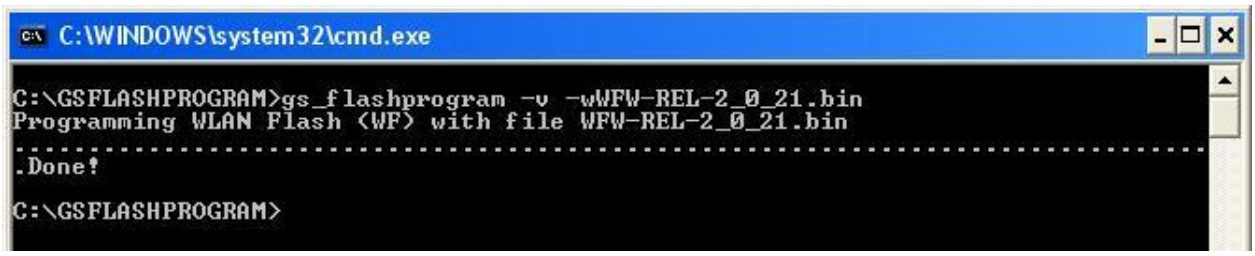



**Figure 23 Loading WLAN firmware**

7). Since the flash is erased, the MAC ID needs to be re-loaded. Customer can either use the GainSpan MAC ID printed on the module label or use their own.


## 3.2.2 Programing MAC ID

1) To Load MAC ID into the flash at the factory default section, type the command ***gs_flashprogram -m mac_address (12 digit hex string)*** This will program MAC ID into the module in the factory default area.

## 3.2.3 Loading APP firmware

Load APP firmware files. The APP files are split into two files, APP1 and APP2. The APP1 file is loaded into the APP FLASH 0 and APP2 file is loaded into the APP FLASH 1.

Type the command below

***gs_flashprogram.exe -v -0<App1 filename>.bin -1< App2 filename>.bin*** as shown in Figure 24

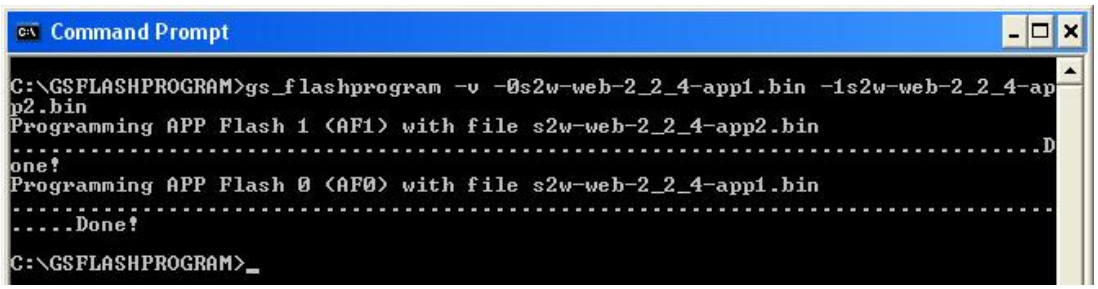



**Figure 24 Loading APP firmware**

This will load the APP firmware. Once this is done, put the module back in run mode, by making sure GPIO27 is no-connected and power cycle the board.

Confirm that the firmware is flashed correctly. If using S2W, you can connect via UART and using a terminal SW such as Tera-term, verify the version by doing AT+VER=? and verify the MAC ID by using command AT+NMAC=?.

## 3.3 Updating Firmware without Erasing Flash Factory Defaults

If Serial to Wi-Fi firmware is already loaded on the module, and customer is updating/upgrading to a newer version of Serial to Wi-Fi firmware, then customer should update without doing a flash erase. This will preserve the contents (MAC ID, etc) in the factory default area that is already programmed in the module and does not need to be reprogrammed.

1). In order to update firmware without erasing flash type command

***gs_flashprogram.exe -v -0<App0 filename>.bin -1< App1 filename>.bin –w<WLAN file name>.bin***

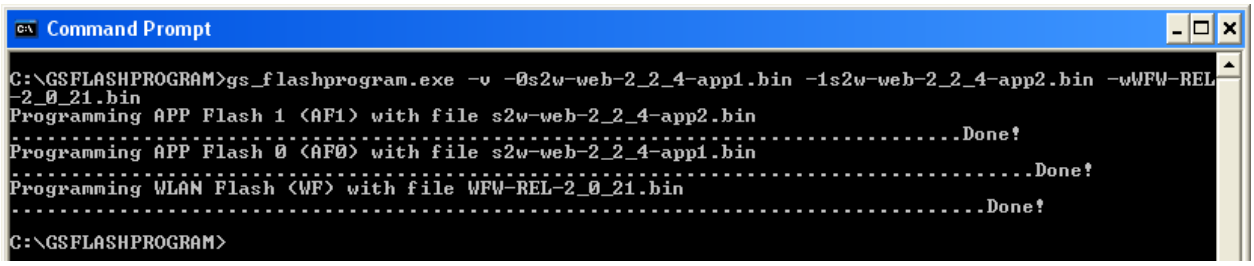



**Figure 25: Updating firmware without erasing FLASH**