Implementation of a Protocol for Booting, Programming and Interfacing of an Indigenous Multichannel Data Acquisition and Processing System

Abey Rose J, Anjana S J, Joji Daniel, Nimmy Joseph, Dr. Deepu Roy, Syamlal L S, Padmakumar K, Jayalekshmy L

doi: https://doi.org/10.37285/bsp.SACAD2025.16

Abstract— **The Multifunction IC (MFIC) is an indigenously designed System on Chip (SoC) that caters to a range of applications, including Navigation, Guidance, Control, Telemetry, Data Acquisition, and Digital Signal Processing. At its core, the MFIC features a flight-proven 32-bit indigenous processor core and a comprehensive System Software toolset that supports programming in both assembly language and high-level languages. The SoC's architecture comprises multiple functional modules with analog interfaces and digital processing cores, which are configured through memory-mapped peripheral registers to operate in continuous mode with minimal processor intervention. To optimize the utilization of external pins for specific applications and to constrain the MFIC within a particular form factor, a requirement has been identified to store the boot code in serial FLASH and subsequently load it into the internal Static Random-Access Memory (SRAM) of the MFIC upon power-on reset.**

To facilitate this functionality, a protocol has been designed and developed to enable communication with an external checkout system via UART. The communication protocol relies on a Load, Read, Go sequence implemented in the boot loader, which incorporates appropriate handshake logic in both the boot loader and the UART hardware controller. This paper provides an in-depth explanation of the boot loader's capabilities, its implementation aspects, conversion to firmware format, and validation process.

Index Terms— **MFIC, boot loader, UART, SPI, FLASH**

I. Introduction

The Multi-Functional Integrated Circuit (MFIC) is designed as a multichannel data acquisition and processing system, integrating signal conditioning, data conversion, and digital processing functionalities. The optimal form factor of the MFIC has been achieved by maximizing the external interface pins for various application domains. However, to conserve pin count, the address and data pins for memory are not brought outside the chip. Instead, the design incorporates a Serial Peripheral Interface (SPI) FLASH, which enables loading of the application code, including boot software, into the MFIC's internal Static Random-Access Memory (SRAM).

Upon power-on, the indigenous 32-bit processor core within the MFIC executes from a fixed start address stored in the 0^{th} memory location. The boot loader, a permanent firmware component, is fused in the first 1k words of memory, starting from this fixed address. As the first program to execute on power-on or external reset, the boot loader implements specific functionalities based on the configurations of the Boot_Config_Pin and Program_Pin.

The boot loader supports various modes of operation, including User Program mode, which enables booting from an external memory device where the user program resides. This is achieved by configuring the multiplexed pins of the MFIC with Data and Address lines. For all internal memory execution modes, the output pins are multiplexed with IO functionality, providing flexibility in system configuration.

Boot_Download mode is the default mode of a system realized with MFIC. MFIC executes from Internal SRAM if boot code is present in memory. Otherwise, the boot loader downloads the boot code from serial SPI FLASH to internal SRAM and transfers control to it. In Boot_Program mode, it supports programming of MFIC without any external programmer hardware. In this mode, the chip executes the protocol for writing to MFIC Internal SRAM. It supports reading data from either MFIC Internal SRAM or SPI Flash, and also, executing an application loaded in MFIC Internal SRAM. User interaction is through the Universal Asynchronous Receiver-Transmitter (UART) interface adhering to RS232 serial communication protocol.

This paper describes the protocols defined and implemented for communication with the UART hardware controller and the functionalities achieved by the boot loader with associated error-handling logics. Due care is given to design the boot loader with a minimal memory footprint and boot download time for the MFIC. The paper also explains the process of converting boot loader software to firmware. It concludes with the different levels of validations carried out to verify the features of the boot loader.

II. Related works

In the existing computer packages designed with the indigenous microprocessor, boot code is developed to support the ‘Load’, ‘Read’ and ‘Go’ protocol through 1553 interfaces. In those designs, the boot code used to be written to an EEPROM using an external utility tool, which was then interfaced with the processor.

As discussed in the references [1], [2], and [3], a boot loader can be integrated into the system to facilitate programming

through an external interface, such as CAN or UART, and to write the boot code to FLASH memory. The boot loader discussed in this paper loads the boot code into the internal SRAM of MFIC and then transfers control to it, enabling the execution of the boot code.

III. COMMUNICATION PROTOCOL BETWEEN MFIC AND SPI FLASH

In Boot_Program mode, processor self test is performed initially. Boot program can then be loaded to MFIC Internal SRAM and a RAM-loadable utility can be used for programming of external SPI FLASH memory device. Protocol is developed to load program from checkout to internal SRAM by transferring data through UART interface, with provision for Cyclic Redundancy Checksum (CRC) verification of entire data. Similarly, data can also be read from Internal SRAM/SPI FLASH to checkout system through UART interface. Protocol also supports execution of a program from a particular address.

A. Handshaking

Load, Read, Go Protocol is implemented in boot loader. Handshaking logic between User and MFIC is implemented in both UART hardware controller and boot loader. It establishes the asynchronous timing on which a command can be given to MFIC and the last command status.

Data/Control command about to be placed on the UART data bus is intimated through UART command 00000084_h placed on the data bus. The last 5 bits represents the number of words to be read, maximum supported being 32 words. Data is then placed on the bus which is sequentially read by the UART hardware controller and is placed on the UART buffer for further processing.

Status register available on the UART hardware module communicates the MFIC processing status to the user. User is expected to issue the next command when MFIC is FREE. Status can be queried using command word 00000040_h. On getting a UART command for Read, UART controller modifies the LS16 bit of Status register to signify that it is in BUSY mode to signal the user that a control command is under processing by MFIC.

On a reset while MFIC is in Program mode, it polls on UART 'Read on Clear' Transfer status register to process the command that is placed on the UART buffer. It changes the LS 16 bits of UART status register to FREE and status of Last command is returned in MS 16 bits after processing the command, denoting that the control command is processed and MFIC is ready for next command.

This handshake logic is part of every command processing. Further processing of Load, Read, Go protocol is done by the boot loader based on the protocol control commands available in UART buffer. All the protocol control commands comprise 4 words as explained subsequently.

B. Protocol Definition

Protocol Control commands received through UART determine the behaviour of boot loader. Each control command is of length 4 words. First word is the Command word and is sent as Command + Command Bar code as below. The list of Commands is provided in the table.

Sl. No	**Command Name**	**Command Code**	**Command Bar _Code**
1	Load Ctrl	0101_h	$FEFE_h$
2	Load Data	0202_h	$FDFD_h$
3	Read Ctrl	0303_h	$FCFC_h$
4	UART Transmit Command	Hardware Command - 0000_h (for 32 words)	
5	GO Control	0707_h	$F8F8_h$

The control commands (Load Ctrl, Read Ctrl, Go Ctrl) are identified based on the first word placed on the UART buffer. For transferring the data from UART buffer to external world through the bus, UART hardware transmit control command is used. For transferring data from external world to MFIC, Load Data format is used where data command is used for identifying the command.

1) Load Control Command

Load Control command is used to send a Control command from Checkout to MFIC for initiating data transfer from Checkout to MFIC Internal SRAM through UART.

Load Ctrl Command shall be sent in the following format to UART Receive Data Buffer (1k*32):

31..16	15..0
CMD BAR	CMD
CRC Integrity Check Not Enabled (0000) / Enabled* - (0001)	0000
MFIC Internal SRAM START ADDRESS	
NO. OF 32-bit DATA WORDS TO BE LOADED	

2) Load Data Command

Load Data command is used to send data from Checkout to MFIC Internal SRAM through UART. A total of 28 Data words are expected in the Load Data command even if less no. of words is to be really transmitted, with Checkout filling the remaining words with 0.

It shall be sent in the following format to UART Receive Data Buffer (1k*32):

31..16	15..0
CMD BAR	CMD
MESSAGEID	
TOTAL MESSAGE COUNT	
XOR CHECKSUM	
DATA WORD 1	
DATA WORD 2	
…..	
DATA WORD 28	

Xor Checksum for Datawords 1 to 28 needs to be computed and sent which will be verified by the Boot loader and success/error codes are set accordingly.

3) Read Control Command

Read Control command is used to send a Control command from Checkout to MFIC for initiating data transfer from MFIC Internal SRAM/SPI FLASH to Checkout through UART.

Read Ctrl Command shall be sent in the following format to UART Receive Data Buffer (1k*32):

31..16	15..0
CMD BAR	CMD
0000	MFIC Internal RAM (0001) / SPI FLASH (0002)
Start Address (Byte Address for Flash/Word Address for MFIC)	
NO. OF DATA WORDS TO BE READ (32 bit)	

4) UART Transmit Command

UART Transmit command is a UART hardware command used to transfer data from UART Transmit buffer to Checkout through UART. UART Transmit Command for transmitting 32 words shall be sent as 00000000 in the following format from Checkout.

Data will be available in UART Transmit buffer (1k*32) in the following format and will be transferred on UART Transmit Command:

MESSAGEID
TOTAL MESSAGE COUNT
XOR CHECKSUM
DATA WORD 1
DATA WORD 2
…..
DATA WORD N

5) GO Control Command

Go Control command is used to send a Control command from Checkout to MFIC for transferring Program Counter (PC) of MFIC to the Address specified in Control Command.

Go Control Command shall be sent in the following format to UART Receive Data Buffer (1k*32):

CMD BAR	CMD
*****	
MFIC GO Address	
*****	

C. Status Codes

Boot loader updates Status codes for every command that is executed. Once command execution is completed it updates MFIC status to FREE along with the last command status. Some of the Success and Error codes and error handling is explained.

a.Success Codes

02FD Load Control Command Protocol handled successfully
04FB MFIC in BOOT_PROGRAM mode after POWERON/RESET. Ready to receive control commands
07F8 PC Transferred to 80000_h (Busy)
08F7 PC Transferred to Go/Execute Address
14EB Load data MFIC Protocol Intermediate Messages handled successfully
15EA Load data MFIC Protocol Last message handled successfully
33CC SPI Flash Sector Erase Success
34CB Block Transfer - Data Transfer & Pattern Writing Completed

Additional success codes are defined for Booting status and Read Message data status.

b. Error Codes

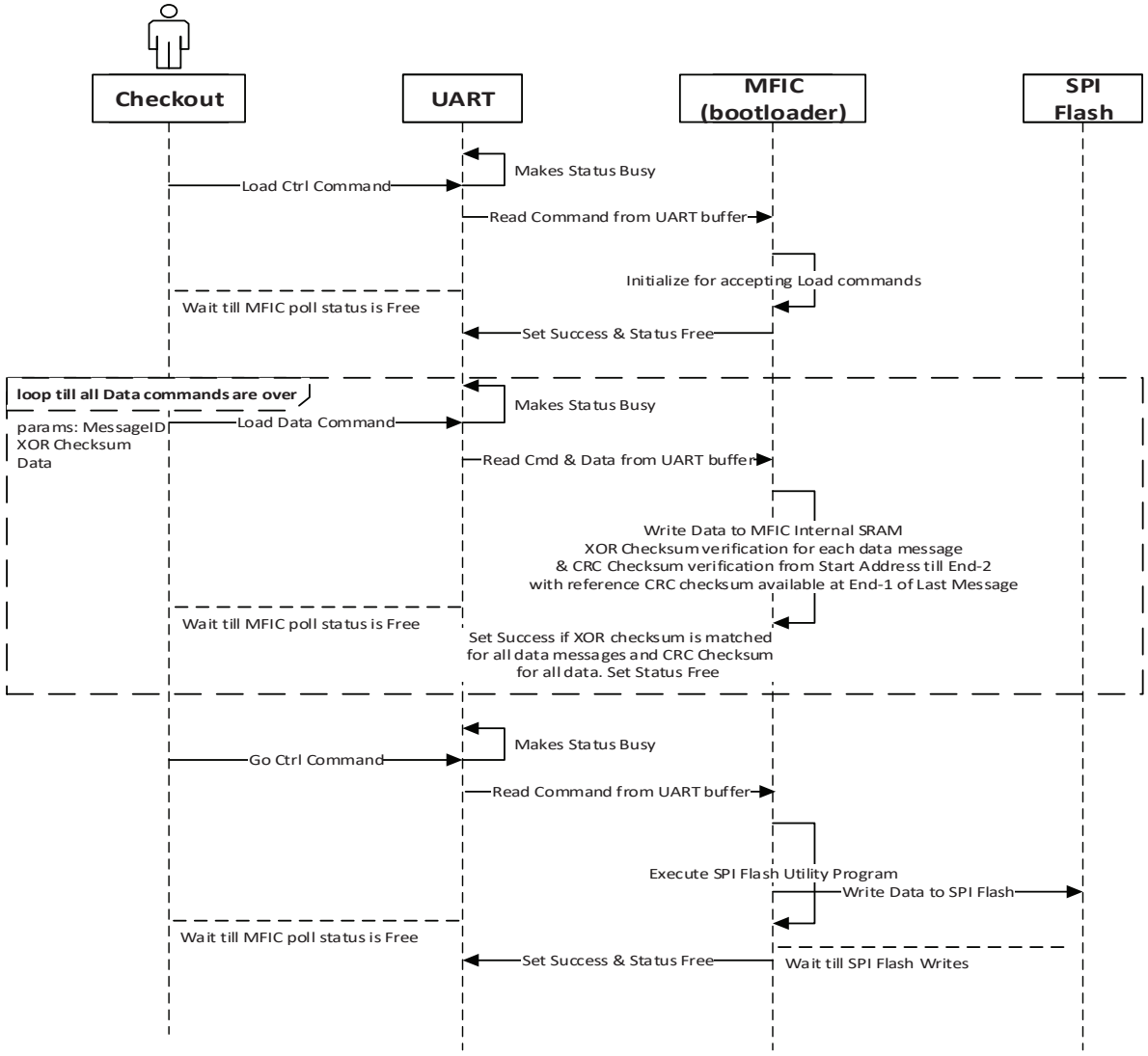
FB04 READ/LOAD/GO Control Command not received as per defined Protocol
FA05 MFIC Internal SRAM (1)/SPI FLASH (2) Indicator of Read Control Protocol out of range value
F609 Address Range of Load Protocol for MFIC Internal SRAM is outside the range of Internal Code SRAM / Internal Data SRAM & Peripheral RAM / System RAM
F40B CRC Integrity Check Enabled (1)/Not Enabled (0) of Load Control protocol out of range value
F30C CRC not matching for Boot loaded from SPI flash Location 0
F20B CRC not matching for Boot loaded from SPI flash Location 20000
EB14 Load Data Command not received as per defined Protocol
EA15 Load Data Protocol Message ID not as expected (computed from Control message) not as expected
E619 Load MFIC data last message XOR not matching
E51A Load MFIC data intermediate messages XOR not matching
E21D CRC Integrity Check in MFIC Internal SRAM failed
DC23 Not able to read from SPI FLASH
C639 Transmit command not received after Read Control Protocol command
A15F Processor Self-test instruction failed
BE41 Block Transfer - Flash Data Readback error
BD42 Block Transfer - Flash Data BOOT_OK Pattern Readback Error
BC43 Sector Erase Failed
BB44 Sector Erase Readback Failed
DD22 Not able to Write on SPI FLASH

There are additional error codes for Boot Download status, and Load & Read command formats.

IV. PROGRAMMING PROCESS

Programming of MFIC is implemented using the above protocol. Commands are sent by checkout based on the Status code received after every command. SPI FLASH commands for reading are compatible across the devices, whereas a considerable difference exists across the devices while writing to SPI FLASH. It necessitated a different approach for writing and reading from SPI FLASH.

A. Sequencing Diagram- Loading Boot Code & Utilities to Internal SRAM and Writing Boot Code to SPI Flash

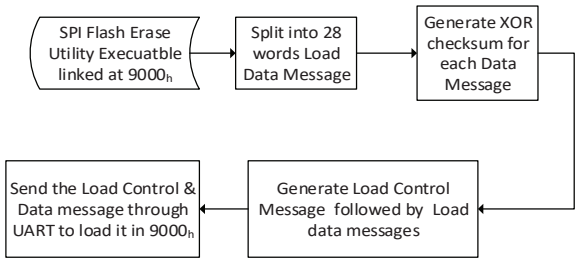


B. Procedure for Loading Boot Code & Utilities to Internal SRAM and Write Boot Code to SPI Flash

For writing Boot code to SPI FLASH, an executable targeting the device specifications can be loaded into MFIC dynamically and executed to copy the Boot code from Internal SRAM to SPI FLASH after erasing the device sector.

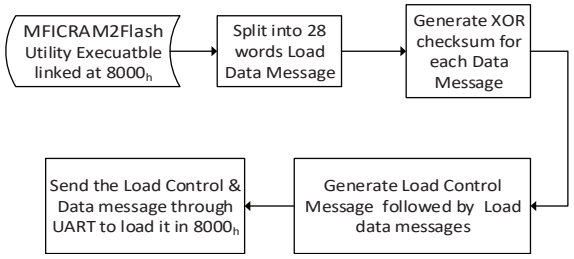
1) Loading SPI FLASH Erase Utility to Internal SRAM

SPI FLASH erase procedure can be different as the commands are different across devices and utility corresponding to the make of SPI FLASH needs to be loaded.



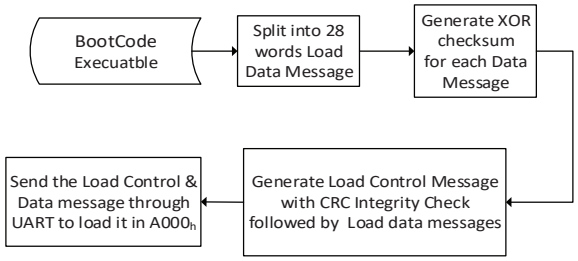
2) Loading SPI FLASH Write Utility to Internal SRAM

Boot code that is loaded to Internal SRAM needs to be written to SPI FLASH. This procedure can be different as the command and word alignments are different across devices. So, write utility corresponding to the FLASH device is to be loaded to MFIC Internal SRAM.



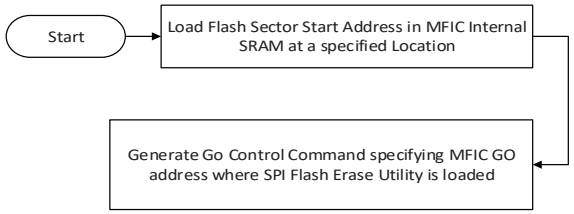
3) Loading Boot code to Internal SRAM

Boot code that is to be written to SPI FLASH has to be loaded into Internal SRAM.



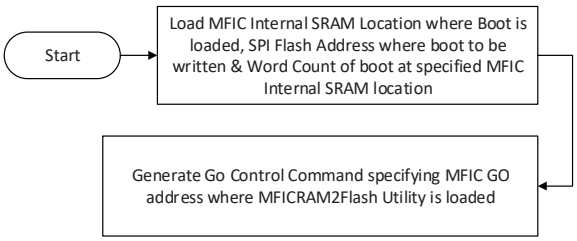
4) Executing SPI FLASH Erase Utility

Erasing SPI FLASH is the first step for writing data into SPI FLASH. It has to be executed after specifying the sector start address as an initialization data.



5) Executing SPI FLASH Write Utility to write Boot code in Internal SRAM to SPI Flash

Writing boot code from Internal SRAM to SPI FLASH is done by executing SPI FLASH Write Utility. It has to be executed after specifying the source start address of boot code in Internal SRAM, word count and target SPI FLASH byte address as an initialization data.



V. BOOT DOWNLOAD PROCESS

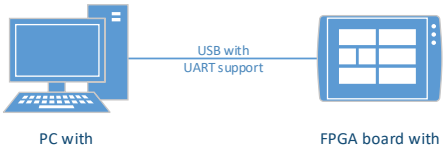
On MFIC reset, Boot loader is executed which checks for the mode based on BOOT & PROGRAM pin configurations of the SoC. In Download mode, Boot loader checks for valid pattern in the SPI FLASH location (00000_h). If valid pattern is present, CRC computation using CRC hardware module is enabled for the boot code downloaded from SPI FLASH into the internal SRAM address 00000_h. Computed CRC is verified against the CRC reference checksum pre-computed by the indigenously-developed linker and stored in a designated location within the boot code. In the event of CRC verification error, a duplicate copy stored in SPI FLASH from the location 20000_h is downloaded. Program control is henceforth transferred to the start address of boot code.

VI. CONVERSION OF BOOT LOADER TO FIRMWARE

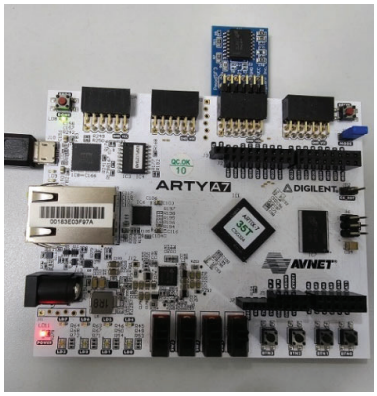
The boot loader developed in assembly language of the indigenous processor core is translated to binary file using Assembler & Linker, both developed in-house. These binaries are fused as part of the source code of MFIC ASIC in Hardware Definition Language (HDL). The conversion from binary format to HDL is performed automatically using a utility program to reduce the chance for errors. The necessary error handling to detect overflows & other incorrect values are also included. This utility is also developed in-house and validated before use.

VII. VALIDATIONS AND RESULTS

Boot loader is simulated both in VHDL Simulator and in a proto-FPGA board along with UART, SPI and processor core. The entire process outlined in sections III & 1V of this paper is verified against the expected status codes implemented in Boot loader.



(a)



(b)

Fig 1: (a) MFIC Boot loader test setup; (b) FPGA board

PC-based Checkout application is developed with UART interfaces for Load, Read & Go functionalities and provision for loading initialization values of the utilities at the specified address.

For loading the data and code files to MFIC Internal SRAM, it is split into message packets of 28 words each as explained in section II.B.2 of this paper. The Load control message as explained in section II.B.1 is generated thereafter and transferred through UART interface, followed by Load data messages.

Checkout application has provisions for reading the code and data from SPI FLASH or Internal SRAM using the Read Control protocol as explained in section II.B.3 and Transmit command as explained in section II.B.4 of this paper. It also has provision for reading the status using the Status command to check the last command status.

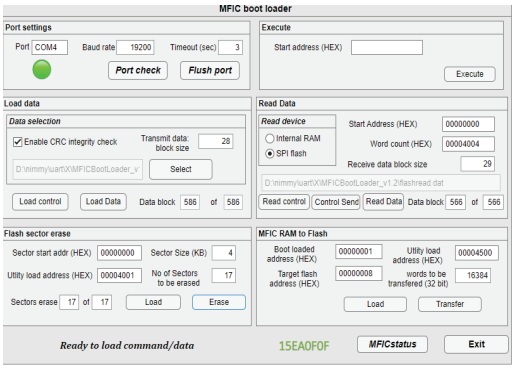


Fig 2 Snapshot of Checkout during Boot code Transfer to MFIC Internal SRAM

The validation outputs are as per the behavior specifications. The error paths implemented in the boot loader are also verified and found satisfactory. For all error cases LED in the proto-board is also lighted up.

VIII. Conclusion

Boot loader software was designed and developed to meet the requirements of programming the MFIC without any external hardware by storing Boot code on Serial SPI Flash and then downloading and executing it on subsequent power-on or external Reset. It was converted to firmware format and fused in the hardware. Different modes of operation of the boot loader were tested on FPGA prototype board interfaced with SPI FLASH. The performance of the boot loader is as expected in the nominal path and error scenarios.

Acknowledgment

We would like to thank the inputs and technical support received from the hardware design team of MFIC and the review committee. We also thank the hardware team for the simulation support provided for verification of boot loader. The authors gratefully acknowledge DD, VSSC (AVN) and Director, VSSC, for allowing the work to be published.

References

[1] Daniel Bogdan, “Design and implementation of a boot loader in the context of intelligent vehicle systems” in *2007 IEEE Conference on Technologies for Sustainability*.

[2] C. Sha and Z.-Y. Lin, “Design Optimization and Implementation of Boot loader in Embedded System Development,” Proceedings of the International Conference on Computer Science and Applications (CSA), 2015.

[3] Application Note, “Guidelines for designing a UART bootloader protocol for STM32WL3 MCUs”