ADSBee 1090 Firmware Reference Guide

Notes about how the firmware works and why.

John McNelly

john@pantsforbirds.com

1 Flash Layout and Programming

1.1 Overview

The ADSBee 1090 includes two microcontrollers with independent flash storage: an RP2040 and ESP32. The system is designed with the RP2040 as the master, and the ESP32 as an optional network controller that can be enabled or disabled by the RP2040. This is intended to support future cost-down and lightweight designs that may omit the ESP32 entirely in favor of a local-only ADS-B decoder solution.

In order to simplify the firmware update process, firmware is flashed to both microcontrollers via a single firmware file (often as a .uf2). Firmware for the RP2040 and the ESP32 is bundled into a single image, and flashed to the RP2040 via USB DFU. On startup, the RP2040 queries the firmware version on the ESP32 via the inter-processor communication SPI bus, and if it finds a value that doesn't match its own firmware version (or no value at all), it re-flashes the firmware on the ESP32 with its bundled ESP32 firmware image.

The RP2040 on the ADSBee 1090 uses an external flash chip with 16MB of storage capacity. The ESP32-S3 on the ADSBee 1090 includes 8MB of storage capacity on PCBA Revision H, but may be reduced to 4MB of capacity on future revisions, since only around 4MB of storage is allocated to a given version of the ESP32 firmware on the RP2040's flash (see flash layout section below for details).

1.2 Flash Layout

The flash layout of the 16MB of memory attached to the RP2040 is shown below. Total flash usage is 16384 Bytes (16MB).

Start Address	Size (kBytes)	Region	Note
0x10000000	176	FLASH_BL	Bootloader
0x1002C000	4	FLASH_HDR0	Application 0 Header
0x1002D000	8096	FLASH_APP0	Application 0 Data
0x10815000	4	FLASH_HDR1	Application 1 Header
0x10816000	8096	FLASH_APP1	Application 1 Data
0x10FFE000	8	FLASH_SETTINGS	Settings

The RP2040's flash memory is erasable in 4kB pages (4096 Bytes), and programmable in 256 Byte sectors. The flash layout for the ADSBee 1090 uses a dual image structure in order to facilitate OTA updates. While the firmware in the APP1 partition is being updated, the firmware in the APP0 partition is running (and is the one doing the update). Each firmware partition is preceded by a 4kB header, which takes the following form.

Application Header Version 0x0

Offset from Start of Header	Bytes	Type	Contents	Note
0x0	4	uint32_t	MAGIC_WORD	Beginning of header magic word. 0xFFFFFFFF: No firmware image has been flashed. Do not read this sector. 0xAD5BEEE: A header and firmware image are present and can be validated for booting.
0x4	4	uint32_t	HEADER_VERSION	Header version number.
0x8	4	uint32_t	LEN_BYTES	Length of application data, in bytes.
0xB	4	uint32_t	CRC	CRC32 of application data.
0xD	4	uint32_t	STATUS	0xFFFFFFFF (BLANK): Firmware partition has been erased, new header has not been written yet. 0xFFADFFFF (VALID): Firmware partition has had its checksum verified (or overridden by a debugger that knows that it's valid, despite the checksum not matching), and is bootable. 0xDEADFFFF (STALE): Usable, but try the other image first. 0xDEADDEAD: Do not use this firmware image.

1.3 Bootloader

The bootloader checks which firmware image(s) are valid and bootable by computing a CRC32 across each available firmware image, and then booting to the best available option.

The bootloader uses the MAGIC_WORD field of each application header to check whether the firmware image and header have been flashed, or if the partition is blank. Headers starting with 0xAD5BEEE indicate that a firmware image and header have been flashed to the partition, while headers starting with 0xFFFFFFFFFFFF indicate that the partition is empty and should not be validated.

Each header includes a STATUS word that can be used to indicate its boot priority. By default, freshly flashed firmware images come with a STATUS word of 0xFFADFFFF, indicating that it is the newest firmware image. When a firmware image is used to flash a new image onto the complementary firmware partition, it marks itself as stale by setting its own STATUS word to 0xDEADFFFFF. This indicates to the bootloader that this firmware image is still valid and bootable, but the other image (with STATUS word 0xFFADFFFF) should be preferred during boot.

A firmware upgrade can be rolled back by marking the new firmware image (the one to be rolled back) with a status word of 0xDEADDEAD. The bootloader will refuse to boot a firmware with a STATUS word of 0xDEADDEAD, and will boot the previous firmware version (with STATUS word 0xDEADFFFF) instead.

1.4 OTA Updates

Over The Air (OTA) updates provide an alternative to USB DFU updates, allowing the ADSBee 1090 to be updated via one of its console interfaces without the need to access the BOOT button or exposed test point in order to put the device into USB DFU mode as a mass storage device.

NOTE: If you have physical access to the ADSBee 1090, a USB DFU operation is usually faster and simpler than an OTA update. OTA updates are intended for embedded applications or receiver installations where the ADSBee 1090 needs to have its firmware upgraded remotely.

1.4.1 .ota File Structure

OTA updates are conducted by sending the binary contents of a .ota file over the serial console, either directly on the USB CONSOLE interface or via the network console interface.

Due to the RP2040 not supporting automatically compiled position-independent-code, the application binary is linked separately for each available application slot in flash, and the .OTA file contains an array of headers and applications, one for each possible application slot. During an OTA flashing operation, the master device that is flashing the file first queries the ADSBee for which linked version of the firmware to send, and then sends the header and application data for the relevant partition.

.ota File Structure

Section	Length	Description
NUM_APPS	4 Bytes	Number of applications contained within the file (for now, just 2).
OFFSETS	4*NUM_APPS Bytes	Offsets (in Bytes) of each header section from the beginning of the file. One 32-bit word per offset, in order from first header to last header.
HEADER_0	20 Bytes (Version 0x0)	Contains application header information as described in the Application Header Version 0x0 table.
APPLICATION_0	LEN Bytes	Binary application data. LEN is the length of the application binary as described in the HEADER_0 section.
…		
HEADER_NUM_APPS	20 Bytes (Version 0x0)	Contains application header information as described in the Application Header Version 0x0 table.
APPLICATION_NUM_APPS	LEN Bytes	Binary application data. LEN is the length of the application binary as described in the HEADER_NUM_APPS section.

1.4.2 OTA Update Sequence

Direction	Text	Note
Send to Console	AT+OTA=GET_PARTITION\r\n	
Console Reply	Partition: 1\r\n	
Send To Console	AT+OTA=ERASE\r\n	Begins erasing the complementary application flash sector and enables the sending of DATA blocks containing raw binary.
Console Reply	Erasing Partition 1.\r\n <status messages> OK\r\n	When an “OK’ is received, it indicates that the complementary application flash sector (including header) have been erased and are prepared to receive an OTA update. ERROR reply indicates that something went wrong and the OTA process should be aborted.
Send to Console	AT+OTA=WRITE, <offset (base 16)>, <len_bytes (base 10)>, <crc (base 16)>\r\n	Writes a chunk of firmware to the complementary flash sector. Chunks can be up to 38400 Bytes long.
Console Reply	READY	
Send to Console	<DATA>	ADSBEE will block until it either receives the number of promised Bytes, encounters an error, or times out. ERROR indicates the chunk did not write successfully, and needs to be erased and re-written.
Console Reply	Verifying with CRC=0x05f3e0e2\r\nOK\r\n	Chunk was written successfully.
…		Write and verify cycle repeats until full image has been written.
Send to Console	AT+OTA=VERIFY\r\n	Trigger verification of the complementary firmware partition that was just flashed.
Console Reply	Verifying partition 0: 20096 Bytes, status 0xFFFFFFFF, application CRC 0x936e5ab2\r\n OK\r\n	OK indicates successful verification, ERROR otherwise.
Send to Console	AT+OTA=BOOT\r\n	Optional, forces a boot into the new firmware immediately. Note that it may take around a minute to get to a stable console interface again, as the ESP32 firmware will be automatically upgraded.
Console Reply	Booting partition 0…\r\n	

2 Inter-Processor SPI Communication

2.1 SPI Packet Definitions

This section details raw packet definitions as can be observed on the SPI bus between the ESP32 and RP2040.

Inter-processor SPI communication is done with maximum transfer lengths of 4096 Bytes. Transfers of large objects like the Settings struct (up to 8kB) are automatically split into multiple smaller transfers.

2.1.1 Master (RP2040) to Slave (ESP32)

The RP2040 writes to the ESP32 in order to change configuration parameters and pass along transponder packets that it has received. The RP2040 also has the ability to read from the ESP32 in order to verify that changes have been properly executed and to perform watchdog functionality.

Master Single Write to Slave

Transfer 1	Master Write Packet					
Byte	0	1	2:3	4:5	6:(n-2)	(n-1):n
MOSI	CMD kCmdWriteToSlave 0x01	ADDR	OFFSET	LEN (unused, since length can be inferred from clocks)	DATA	CRC
MISO						

Master Single Read from Slave

Transfer 1	Master Read Request packet				
Byte	0	1	2:3	4:5	6:7
MOSI	CMD kCmdReadFromSlave 0x03	ADDR	OFFSET	LEN	CRC
MISO					

Handshake line goes HI.

Transfer 2	Slave Read Response Packet		
Byte	0	1:(n-2)	(n-1):n
MOSI			
MISO	CMD kCmdDataBlock 0x07	DATA	CRC

Handshake line goes LO.

2.1.2 Slave (ESP32) to Master (RP2040)

The ESP32 writes to and reads from the RP2040 in order to request settings data (only the RP2040 has access to EEPROM), and to pass along data received from its network connection (e.g. network console commands, firmware updates, etc).

Slave Single Write to Master

Handshake line goes HI.

Transfer 1	Slave Write Packet					
Byte	0	1	2:3	4:5	6:(n-2)	(n-1):n
MOSI						
MISO	CMD kCmdWriteToMaster 0x04	ADDR	OFFSET	LEN	DATA	CRC

Handshake line goes LO.

Slave Single Read from Master

Handshake line goes HI.

Transfer 2	Slave Read Request Packet					Master Read Response Packet		
Byte	0	1	2:3	4:5	6	7	8:(n-2)	(n-1):n
MOSI						CMD kCmdDataBlock 0x07	DATA	CRC
MISO	CMD kCmdReadFromMaster (0x06)	ADDR	OFFSET	LEN	CRC			

Handshake line goes LO.

2.1.3 SPI Exchange Examples

Below is a non-exhaustive list of example transactions illustrating how some of the SPI packet definitions are used for communication between the RP2040 and ESP32.

2.1.3.1 RP2040 writes small object to ESP32 without ACK

- RP2040 asserts chip select.
- RP2040 sends single transfer with CMD = kCmdWriteToSlave.
- RP2040 de-asserts chip select.

2.1.3.2 RP2040 writes small object to ESP32 with ACK

- RP2040 to ESP32 single write
    - RP2040 asserts chip select.
    - RP2040 sends single transfer with CMD = kCmdWriteToSlaveRequireAck.
    - RP2040 de-asserts chip select and waits for handshake.
- ESP32 to RP2040 handshake
    - ESP32 asserts HANDSHAKE line.
    - RP2040 asserts chip select.
    - RP2040 reads the first byte of the incoming message to determine that it's an ACK. If the message is an ACK, the transaction was acknowledged and succeeds, otherwise an error is thrown.
    - RP2040 de-asserts chip select.
    - ESP32 de-asserts HANDSHAKE line.

2.1.3.3 RP2040 writes large object to ESP32 with ACK

- RP2040 breaks large object into multiple single transfers, each of which gets sent in the same manner as "RP2040 writes small object to ESP32 with ACK". The offset for each transfer gets incremented based on the starting address and size of the previous chunk that was successfully sent.

2.1.3.4 ESP32 reads small object from RP2040 without ACK

- ESP32 asserts HANDSHAKE line.
- RP2040 asserts chip select.
- RP2040 reads first byte of incoming message and determines that it's a kCmdReadFromMaster. RP2040 reads the address, offset, and length fields and writes its response into the remainder of the packet.
- RP2040 de-asserts chip select.
- ESP32 de-asserts HANDSHAKE line.

2.2 Object Dictionary

Within each SPI packet, object dictionary definitions are used to convert the address, offset, and payload fields into actions that are performed on the RP2040 and ESP32. For instance, the RP2040 can write to the kAddrRawTransponderPacket address via SPI in order to forward a received transponder packet to the ESP32's onboard aircraft dictionary.

Dictionary addresses and their contents are shown below.

Address	Name	RP2040 R/W	ESP32 R/W	Dictionary Contents
0x01	Firmware Version	-	R	Firmware version as a uint32_t. Major, minor, and patch versions are each represented as a uint8_t, so the full firmware version is expressed as follows. `const uint32_t ObjectDictionary::kFirmwareVersion = (kFirmwareVersionMajor) << 16 | (kFirmwareVersionMinor) << 8 | (kFirmwareVersionPatch);`
0x02	Scratch Register	RW	RW	Scratch register (uint32_t) that can be read or written to confirm that serial communication is working.
0x03	Settings Data	RW	RW	Access to the SettingsManager's settings struct. Used to synchronize settings information between the RP2040 and the ESP32. Note that settings are stored in EEPROM which is only accessible to the RP2040, so on startup the ESP32 will query the RP2040 for settings information.
0x04	Raw Transponder Packet	-	W	Raw transponder packet sink for RP2040 to write to in order to send RawTransponderPacket objects to the ESP32 in order for it to maintain its own aircraft dictionary. No longer used in favor of the Raw Transponder Packet Array sink, which consolidates multiple packets together to save bandwidth.
0x05	Decoded Transponder Packet	-	-	Not currently supported, but available for future use in case it makes sense to send decoded transponder packets from the RP2040 to the ESP32 in order to save compute for packet decoding (at the cost of additional bandwidth for the larger packet size).
0x06	Raw Transponder Packet Array	-	W	Sink for Raw Transponder Packets flowing from the RP2040 to the ESP32. Data written to this address consists of a one-byte value for the number of packets being sent, and then an array of RawTransponderPacket objects.
0x07	Decoded Transponder Packet Array	-	-	Reserved for future use.
0x08	Aircraft Dictionary Metrics	W	-	Statistics from the aircraft dictionary (number of demodulations, etc) forwarded from the RP2040 to the ESP32.
0x09	Device Info	R	-	Queries device info (MAC addresses, etc) from the ESP32.
0xA	Console	R	W	Allows the ESP32 to forward network console commands (for network-based control and OTA updates) to the RP2040. These

				commands are interpreted by the RP2040 as if they were entered directly via the USB console. Allows the RP2040 to send AT command replies to the ESP32. Note that other console prints (e.g. informational logs / warnings / errors) are not forwarded to the network console interface in order to preserve SPI link bandwidth.
0xB	Network Info	R	-	Queries ESP32 network information (WiFI access point / WiFi station / Ethernet status, IP address, etc).
0xC	Device Status	R	-	Queries status of pending messages from the ESP32.
0xD	Log Messages	R	-	Downloads log messages from the ESP32.

3 ADS-B Decoder

ooOOoo magic

jk I'll write this up later