ADSBee 1090 Firmware Reference Guide

Notes about how the firmware works and why.

John McNelly

john@pantsforbirds.com

# Flash Layout and Programming

## 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).

## 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 | 8100 | FLASH\_APP0 | Application 0 Data |
| 0x10816000 | 4 | FLASH\_HDR1 | Application 1 Header |
| 0x10817000 | 8100 | FLASH\_APP1 | Application 1 Data |

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. |

## 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.

## 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.

### .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. |

### OTA Update Sequence

|  |  |  |
| --- | --- | --- |
| **Direction** | **Text** | **Note** |
| 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 | 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 |  |
|  |  |  |
| Send to Console | DATA=AAAALLLLECCCCXXXX… | The “DATA=” ASCII prefix indicates that the characters following the “=” sign are in raw binary (not human readable ASCII characters).  AAAA: uint32\_t indicating the address offset from the start of the application header sector where the appended data should be written.  LLLL: uint32\_t indicating the number of raw binary Bytes following the address uint32\_t.  E: Indicates whether a CRC32 checksum will be included. 0x01 if using a checksum for the sector, 0x00 if no checksum is included.  CCCC: CRC32 checksum of the binary data (X’s) in this data block only. 0x00000000 when not in use.  XXXX….: Raw binary bytes. Console will read raw binary data until it receives LLLL bytes or reaches a timeout value. |
| Console Reply | Flashed <length> Bytes at offset 0x<addr>.\r\n  OK\r\n | Human-readable status message indicating the number of Bytes and the location they were flashed to.  OK indicates that the next data block can be sent.  ERROR reply indicates that something went wrong and the OTA process should be aborted. Future firmware versions may support the ability to erase targeted firmware sectors in order to allow flashing of individual sectors to be retried in case they get corrupted. |
| Send to Console | … | Repeat DATA block. |
| Console Reply | … | Repeat DATA reply. |
| Send to Console | AT+OTA=COMPLETE\r\n | Indicates that the OTA flashing process is complete. |
| Console Reply | Validating application image in partition n.\r\n  Calculated CRC: 0xNNNNNNNN\r\n  Header CRC: 0xNNNNNNNN\r\n  OK\r\n | Displays the results of the OTA CRC validation.  OK indicates that the validation is successful. The current firmware image will be marked as stale using its STATUS word in the application header, and the new firmware image will be used upon next boot. |
| Send to Console | AT+REBOOT\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. |

# Inter-Processor SPI Communication

## 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.

### 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  kWriteToSlave | 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 |
| **MOSI** | CMD  kReadFromSlave | 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  kReadFromSlave | DATA | CRC |

Handshake line goes LO.

### 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  kWriteToMaster | 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  kDataBlock | DATA | CRC |
| **MISO** | CMD  kReadFromMaster | ADDR | OFFSET | LEN | CRC |  |  |  |

Handshake line goes LO.

### 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.

#### RP2040 writes small object to ESP32 without ACK

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

#### 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.

#### 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.

#### 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.

## 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). |
| 0xB | 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. |
| 0xC | Decoded Transponder Packet Array | - | - | Reserved for future use. |
| 0x06 | Base MAC | - | R | Allows the RP2040 to query the base MAC address on the ESP32. All default MAC address values are calculated from the base MAC address. |
| 0x07 | WiFi Station MAC | - | R | Allows the RP2040 to query the MAC address that the ESP32 uses while operating as a WiFi station (connected to an existing network). |
| 0x08 | WiFi Access Point MAC | - | R | Allows the RP2040 to query the MAC address that the ESP32 uses while operating as a WiFi access point (hosting its own network). |
| 0x09 | Bluetooth MAC | - | R | Allows the RP2040 to query the MAC address that the ESP32 uses for its Bluetooth interface. |
| 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. |

# ADS-B Decoder

ooOOoo magic

jk I’ll write this up later