**Small Satellite Research Laboratory**
*Franklin College of Arts and Sciences*
UNIVERSITY OF GEORGIA

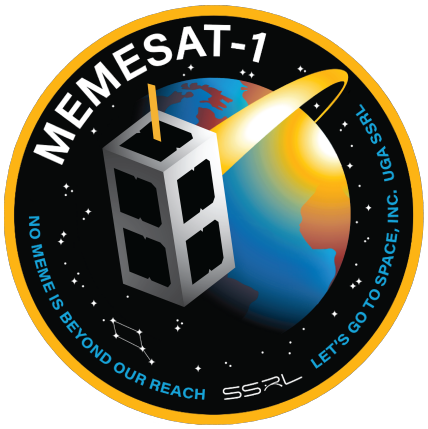
# OTAUpdater Component Documentation

Prepared for
The Small Satellite Research Laboratory
Fall 2024 - Spring 2025

Principal Investigator
Dr. Deepak Mishra

The University of Georgia
Geography-Geology Department
Athens, Georgia
United States of America

Prepared by
The MEMESat-1 CDH Team
University of Georgia
Athens, Georgia
United States of America



# Contents

# 1 List of Acronyms

**BBS** Bulletin Board System
**CDH** Command and Data Handling
**EPS** Electrical Power System
**GDS** Ground Data System
**FSM** Finite State Machine
**FSW** Flight Software
**MCU** Microcontroller Unit
**MS-1** MEMESat-1
**OBC** On-Board Computer
**OSAL** Operating System Abstraction Layer
**OTA** Over-The-Air
**OTAU** Over-The-Air Updater
**TLM** Telemetry
**UART** Universal Asynchronous Receiver-Transmitter
**UHF** Ultra High Frequency

# 2 Revision History

| **Changes** | **Authors** | **Version** |
|---|---|---|
| [04-04-2024] Created document to reflect software design and implementation of the OTAUpdater component. Preparation for IMR-3 | Samuel Lemus | 0.1.0 |

# 3 Introduction

The ability for a flight software setup to be able to change its functionality while in orbit is a critical feature for any satellite. This document outlines the design and implementation of the OTAUpdater component, which is responsible for the periodic tracking of the file system to note when a file has been fully received from a segmented file uplink.

# 4 Purpose

The purpose of the OTAUpdater component is to provide a mechanism for the flight software to change its software configuration while in-flight. In the case where a component would need to be updated, the flight software may need to be rebuilt. As the satellite is prepared, the filesystem contains minimal system packages and bloat. This would lead to there being a single binary of the flight software to be executed on system startup. To update the binary file, we would need to (generally) continue with the following steps.

1. The target source code is recompiled; the binary is generated.
2. The file uplink preparation must have in the groundstation uplink buffer enough space to hold the target binary file.

3. There must be enough bandwidth to send the binary file to the satellite given a 90 second pass-over.
4. The satellite must be in a state where it can receive the binary file (including having enough memory to store it).
5. The satellite takes the data from the file uplink buffer and writes it to the filesystem.
6. The satellite proceeds to execute a power-cycle with the target binary file specified to be ran on startup instead of the previous binary file.

## 4.1 Considerations

### 4.1.1 Scenario 1 - File Too Big for a Single Pass

One of the first concerns sought to be addressed is the case where the target binary file is too big to uplink within 90 seconds given the communications bandwidth. This would result in the scenario where (1) all of the uplink time consumed the pass-over operations for the discrete pass, and (2), there is 90 minutes until the next pass-over and the ground-station file uplink buffer would continue to be filled with the target binary file. This could be a risk for a few reasons.

### 4.1.2 Scenario 2 - Segmented File Uplink Abstraction

With the ability to send a target binary, we may as well consider the capability for the design of the component to be able to tolerate a file of any type. This would not only allow for the capacity for a binary-encoded file to be sent and recreated, but also for long text files or other types of data to be sent and reassembled.

### 4.1.3 Scenario 3 - Target Binary Received and Malformed

In the case where the target binary file is received and is incomplete or malformed, the functions of this component should account for this, discard safely, and log the error in the form of telemetry.

# 5 Interactions

## 5.1 Components

### 5.1.1 Scheduler

The Scheduler component will be responsible for the periodic execution of the OTAUpdater component. The custom rate groups hadn't seemed to be the absolute best solution for the threaded dispatch of this component as it needs to do a considerable amount of file operations.

### 5.1.2 FileRecycler

The FileRecycler component will be responsible for the clean-up of the OTAUpdater component. This will be done by having the OTAUpdater component send a command to the FileRecycler component to delete any files that are no longer needed.

### 5.1.3 StateMachine

The StateMachine component will send a command to the OTAUpdater component to notify that it is scheduled to be powered off or rebooted. Due to the nature of the OTAU's filesystem operations, it is important to ensure that the OTAU is not in the middle of a file operation when it is powered off or rebooted.

#### 5.1.4 Directory Organization (Visual)

```
+-- updater_dir /
|   +-- tracking.txt
|   +-- complete /
|   +-- file_1 /
|   |
|   |   (example: currently building file_1 pre-audit)
|   |   +-- index.txt
|   |   +-- summary.txt
|   |   +-- file_1a /
|   |   |   +-- file_1a.seg
|   |   |   +-- file_1a.md5
|   |   +-- file_1b /
|   |   |   +-- file_1b.seg
|   |   |   +-- file_1b.md5
|   |   +-- file_1c /
|   |   |   +-- file_1c.seg
|   |   |   +-- file_1c.md5
|   |   +-- .. /
|   |
|   |   (example: subdirectory post-audit (still building))
|   +-- file_2 /
|   |   +-- index.txt
|   |   +-- summary.txt
|   |   +-- file_2a.seg
|   |   +-- file_2b.seg
|   |   +-- file_2c.seg
|   |   +-- file_2d.seg
|   |   +-- file_2e /
|   |   |   +-- file_2e.md5
|   |
|   |   (example: subdirectory post-audit
|   |       (done building - awaiting checksum eval))
|   +-- file_3 /
|   |   +-- file_3a.seg
|   |   +-- file_3b.seg
|   |   +-- file_3c.seg
|   |   +-- file_3d.seg
|   |   +-- ..
|   |   +-- file_3.sha256
+-------------------------------
```

# 6 Design

## 6.1 Requirements

| Requirement | Description | Verification Method |
|---|---|---|
| OTAU-001 | The OTAU must be able to track objects in its working directory. | Implementation Test |
| OTAU-002 | The OTAU must be able to resume functions after a volatile memory flush. | Implementation Test |
| OTAU-003 | The OTAU must be able to store and log parameters about intermittently built/building files. | Implementation Test |
| OTAU-004 | The OTAU must be able to control gpio pins which permit the STM to enter bootloader mode. | Implementation Test |

## 6.2 Design Overview

1. OTAU needs to bind to the scheduler to run at a sub hz rate.
2. OTAU needs to connect to the file recycler to clean-up compiled directories.
3. OTAU needs to report (at minimum) the following telemetry:
   (a) n objects building.
   (b) n objects built.
   (c) n objects tracked.
   (d) n times a file difference has been noticed
4. OTAU needs to do the following:
   (a) Track the number of objects in the respective working directory
   (b) Report all objects being tracked in a file at the base of the working directory.
   (c) Ingest the index.txt report parameters of the target file currently being built (held in the respective object directory)
   (d) Save any current audit statistics to a summary.txt file in the respective object directory.
   (e) Identify when a file has been fully received in its segments and compare the culmination against a checksum.
   (f) If the object group satisfies the checksum comparison, then the file is converted to a previously specified target format.
   (g) The target file may either sit in the segment directory until further action from gds controls or have specified a target directory for the file to be moved to.
   (h) In the case of (g), there must also be held in the segmented directory a script labeled 'setup.sh' which prepares the filesystem for the presence of this new file.

(i) The OTAU should retain the ability to flash the STM EPS firmware by 'pulling high' boot (0 or 1) and the resent pin.

i. Q: Should the OTA trigger the state machine to prepare for an interrupt of data coming from the stm?

ii. Q: If the OTAU is to trigger the StateMachine, does it have to do so directly or may it do so my alerting the stm and the stm then triggers the StateMachine/EpsUart.

### 6.3 Ports

| **Port Data Type** | **Name** | **Direction** | **Kind** | **Usage** |
|---|---|---|---|---|
| UpdaterModule UpdateOpPort | updaterOpIn | input | asynchronous | This port is used to receive commands from the ground station. |
| Drv GpioWrite | gpioWrite | output | | Port used to write to the GPIO pins. |
| SchedulerModule SendSchedulePort | sendSchedule | output | | Port for sending the schedule to the scheduler. |
| SchedulerModule RunSchedulePort | scheduledHandler | input | asynchronous | Port for receiving the schedule from the scheduler. |
| FileRecyclerModule CleanUpDataPort | cleanupDataPath | output | | Port for sending a segment directory to be cleaned by the file recycler. |
| StateMachineModule SM_ReadyForPowerOff | readyForPowerOff | input | synchronous | Port for receiving the power off command from the state machine. |

### 6.4 Parameters

| Parameter Name | Type | Default Value | Description |
|---|---|---|---|
| OTA_Schedule | string | ”0 * * * * ?” | Schedule formatting for the OTAU. |
| StmBootOutNum | StmBootOutNum BOOT_0 | | Pin number for the STM boot pin. |
| rstState | StmRstState LOW | | Pin number for the STM reset pin. |
| stmBootTime | U32 | 2000 | Time in milliseconds to wait for the STM to boot. |
| stmRstTime | U32 | 1000 | Time in milliseconds to wait for the STM to reset. |

### 6.5 Commands

| Mnemonic | Arguments | Synchronization | Description |
|---|---|---|---|
| OTA_PerformAudit | dir_path: string | asynchronous | Command to perform an audit of the OTAU. |
| OTA_PerformFileOp | operation: U32<br>file_path: string | asynchronous | Command to perform a file operation on the OTAU. |
| OTA_SelectBootPin | bootOutNum: U32 | asynchronous | Command to select the boot pin for the STM. |
| OTA_SelectRstState | rstState: U32 | asynchronous | Command to select the reset state for the STM. |

### 6.6 Events

| Name | Severity | Arguments | Description |
|---|---|---|---|
| OTA_AuditStatus | warning low | message: string | Event for when the audit reports a state. |
| OTA_AuditComplete | warning low | message: string<br>segments_present: U32<br>segments_missing: U32<br>objects_tracked: U32 | Event for when the audit has been completed. |
| OTA_AuditInterrupted | warning low | message: string<br>status: U32 | Event for when the audit has been interrupted. |
| OTA_FileOperationComplete | warning low | message: string | Event for when a file operation has been completed. |
| OTA_BootPinSelectTrigger | warning high | bootOutNum: StmBootOutNum | Event for when the boot pin has been selected. |
| OTA_RstStateChangeTrigger | warning high | rstState: StmRstState | Event for when the reset state has been changed. |
| OTA_StmRstGetTrigger | warning high | value: StmRstState | Event for getting the stm reset pin state. |
| OTA_InvalidPinCommand | warning low | val: U32 | Invalid pin command requested. |
| OTA_ScheduleError | warning low | status: SchedulerModule Status | Event for when the schedule has an error. |
| OTA_ScheduleRan | diagnostic | | Event for when the schedule has been ran. |
| OTA_ScheduleChangedTo | diagnostic | schedule: string | Event for when the schedule has been changed. |

### 6.7 Telemetry

| Name | Data Type | Update | Description |
|---|---|---|---|
| OTA_ObjectsTracked | U32 | periodic | Number of objects being tracked. |
| OTA_CompleteDormantFileCount | U32 | periodic | Number of files which have been built and do not have a predetermined target directory. |
| OTA_ObjectAdjustmentCount | U32 | periodic | Number of objects which have been adjusted. |
| OTA_SegmentCleanCount | U32 | periodic | Number of segments which have been cleaned by the file recycler. |

### 6.8 Conceptual Control Flow

**Stage #0 → Stage #1**

- Check if `updater_dir` exists
  - If not, attempt to create it
  - Create and evaluate status object
  - Proceed if created; log critical error otherwise
- Attempt to create `updater_dir/tracking.txt`
  - Create and evaluate status object
  - Proceed if created; log critical error otherwise
  - Populate file with base parameters

**Stage #1 → Stage #2 (Audit Process)**

- Read and store directory size
- Create and populate list of directory contents
- Initialize vector: `<path, <param, value>>` for subdirectory data
- For each entry in directory:
  - If file:
    - Should be `tracking.txt` only — log it
  - If `tracking.txt`:
    - Read size, create buffer, parse into key:value vector

- If subdirectory:
  - Initialize subdir vector if not present
  - Read subdir size and contents
  - Check for:
    - `tracking.txt`
    - {`parent_dir`}`.concat`
    - `*.seg` files
  - If no `tracking.txt`: delete subdir
  - If present:
    - Parse and add to main vector
  - If `*.seg` files found: store paths
  - If `.concat` file found, compare file size to target:
    - $<$ target: evaluate segments
    - $=$ target: run checksum
    - $>$ target: delete subdir
  - Else if `*.seg` exist: create concat file

**Segment File Evaluation**

- Open concat file for appending
- Sort segment paths lexicographically
- For each segment:
  - If size $\leq$ target:
    Read into buffer, write to concat, delete segment
  - If size $>$ target:
    Delete segment and any following files; log action
  - Track size written; stop and delete if it exceeds target
- After writing:
  - If size $<$ target: update tracking
  - If size $=$ target: run checksum
  - If size $>$ target: delete subdir

**Checksum Comparison**

- Read concat file into buffer
- Compute MD5 checksum
- Compare to expected checksum:
  - If match:
    Rename, move to `updater_dir/complete`, delete subdir, update tracking
  - If mismatch:
    Log, delete subdir, update tracking