UNIVERSITY OF MARYLAND

G49 - Maintainers Manual

Microprocessors (ENEE440)

Author

Søren Graae

121329035

December 16, 2024

Contents

1 Introduction

The purpose of the Maintainers Manual is to give an overview of the system, enabling a maintainer to adjust it to their needs; add functionality, produce potential fixes, add a new device, etc.

1.1 Abstract

The G49 Micro-on-Tether system is built onto an STM32G491RE microcontroller. The system has a multitude of Devices that allows a user to quickly execute prototypes, develop products that have multiple inputs or outputs, or implement it in a more complex system such as a control panel for automated fabrication lines. This works by sending the system a command string via UART, which is dispatched by a command dispatcher Device.

Built-in Devices include:

- **Digital to Analog Converter (DAC1)**: Device outputs an analog voltage to the userLED pin.
- **Timer (TIM2)**: Device provides the ability to output a consistent digital signal to the userLED pin.
- **Serial Peripheral Interface (SPI2)**: Device provides communication for the W25Q128 flash memory module.

Each Device has executable commands that run once, when executed by the user, and each Device has a multitude of tasks; only one installed. The installed task for a Device is run each time the Device is reached, and by default the installed task has no operations. To return information to the user, each Device can *post* a message via UART.

2 System Overview

The G49 Micro-on-Tether (MoT) system is deployed on an STM32G491RE microcontroller.

2.1 Big Picture

The system is made up of multiple of what is termed a *Device*. These Devices all serve different purposes and consists of commands and tasks. A Device receives a command through the dispatcher, which is a special Device; Device 0.

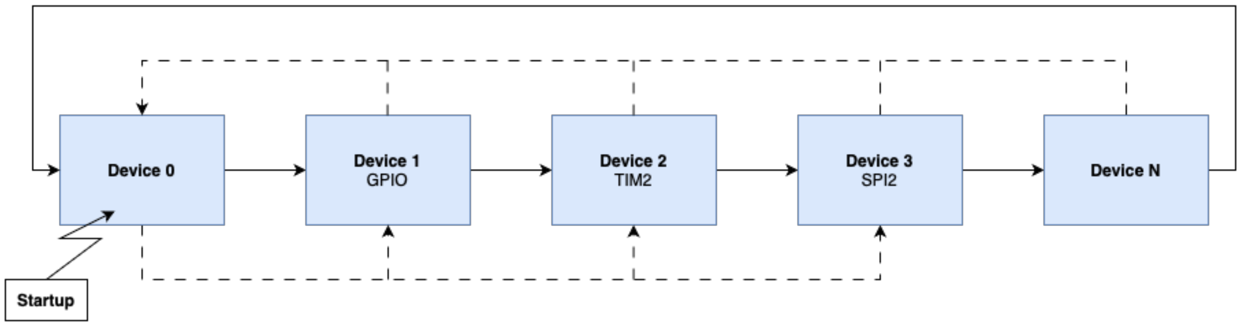

Figure 1: General flow of system

On Figure 1 a *simplified* flowchart of the system is illustrated. The purpose of this is to give a general idea of how the system prioritizes and executes tasks.

The system iterates through all Devices, however, it is important to notice that Device 0 and Device N are special devices. The purpose of Device 0 is to act as a "command-dispatcher", forwarding a command to the given Device. DeviceN handles commands sent from one Device to another Device; inter-Device commands.

A command is like a function - it's executed once and the Device moves on to the next. On the other hand, each Device also consists of at least one, often multiple, tasks that performs a job each time the Device is reached. In chapter 4 Tasks, the usefulness and implementation of a task is explained further.

To communicate with a user, the system can send messages back via the UART connection. This is termed *posting*. The concept is generally simple and will be explained further in chapter 5 Devices.

2.2 Command Strings: Structure and Execution

A command is issued by a *command string*, which consists of four (to five) main parts:

```
:  XX YY (ZZ ...)  WW
```

Each letter group represents a hexadecimal number; `0xYY`.

To initiate the command, a '`:`' colon is transmitted. This lets the dispatcher know that a command has been received. Afterwards one byte of data is transmitted, this is the Device number; `XX`. This tells the dispatcher which Device the command is intended for. The dispatcher "forwards" the command string to that Device, excluding the colon and the Device number. What remains is:

```
YY (ZZ ...)  WW
```

Here, `YY` is the command number. This tells Device `XX` which command has been received. The Device then executes the given command. If necessary, the command string can include inputs - either 8-bit, 16-bit, or 32-bit. This is the `ZZ ...` part of the command string and can be omitted if the chosen command has no input. The length of the data-input simply depends on the memory available on the device, but practically there's no limitation concerning most use cases, which means that `ZZ ...` can be extended as much as needed, e.g.:

`ZZ ...` $\Rightarrow$ `ZZ UUUU PP GGGGGGGG`

In the example above, the input data `ZZ ...` consists of four different inputs: `ZZ` is an 8-bit input, `UUUU` is 16-bit, `PP` is 8-bit, and `32-bit`.

The last part of the command string is `WW`. This is absolutely necessary for the command to be accepted by the system. `WW` is a checksum *byte*, and should make the summation of `XX`, `YY`, and `(ZZ ...)` result in 0x00.

A command string can be generated by hand, although ideally the utility program *h2m* (hex2mot) is used. The reason for this is that the command string is formatted as little endian and the checksum calculation is automated.

	Initializer	**Device**	**Command**	**Input**	**Checksum**
Input		03	01	02 0044	
Output	:	03	01	02 4400	B6

Table 1: Example of command string generation

Looking at Table 1, the generation of a command string can be seen. h2m receives `03 01 02 0044` and outputs the command string `:0301024400B6`. This string is sent to the G49 MoT system that dispatches the command along with the input arguments. From this particular command string, command 1 in Device 3 is executed with input `020044`. The checksum is not used in the Device.

3 Commands

As mentioned previously, when a command is received in Device0, it is dispatched to the given Device along with the input arguments.

```
    .thumb_func
SPI2_INITcmd:
@;  push {r7,lr}    @; done above at command entry, so dont do here!
    str r0, [rDEVP, #SPI2_CMDARGSADDR]    @; Store the address for
       SPI2 arguments
    bl SPI2_init

    add r0,rDEVP,#INITIALIZED_msg   @; r0= address of the 'skip'
       message
    MOV_imm32 r1,consoleMsgs    @; consoleMsgs is defined in
       MoTdevice_device0

    bl MoT_msgPost    @; msgPost is only called by commands or tasks

    @; returns here with r0 == 0 : success, r0 == -1 failed to post.
       return value not used at present
    @; return to command dispatcher
    pop {r7,pc} @; command is completed
```

Listing 1: Example command: SPI2 Initialization

Listing 1 shows the SPI2 initialization command. This is used to explain the concept and structure of a command.

Like any other function, it's first defined. In this case it's a local function, but for debugging purposes it's possible to make it global. There's no requirement for the label of the function, but it's generally seen that `cmd` is used when the function is a command, compared to a task which has `task` at the end.

When the command is dispatched, register `r0` holds the address to the Device arguments. For this Device, the address of the arguments is saved in a variable. This is done to enable access later in a command if `r0` has already been cleared or altered. `rDEVP` is the base of the device, to which an offset is applied to access Device specific variables, buffers, messages, etc...

After the address has been stored, the function SPI2_init is branched to. This function is defined in a Low-Level (LL) Device assembly file. In this case, SPI2_init simply configures the SPI2 peripheral to enable communication with a W25Q128 flash memory module.

As the end of the command is reached, a message can be posted. This is done by adding the address of the message to r0, moving a pre-defined address for `consoleMsgs` to r1, and then branching to MoT_msgPost. For this command, the message offset INITIALIZED_msg is used.

Finally, the stack is popped to conclude the command execution.

3.0.1 Inter-Device Commands

To enable the posting of a Device command from another Device, an additional configuration is needed; a command-link.

```
MoT_cmdLink_m SPI2_WHOAMI_cmd, SPI2_WHOAMI_cmdbin, SPI2_WHOAMI_cmdlen
    ,
SPI2_WHOAMI_cmdbin:
.byte 3,1   @; Device 3 is this device, function 1 is SPI2_WHOAMIcmd
.equ SPI2_WHOAMI_cmdlen, ( . - SPI2_WHOAMI_cmdbin)
.align 1 @;!!important to do this where odd-byte alignment is
    possible
```

Listing 2: Example command-link: SPI2 WHOAMI, I

Listing 2 shows the definition of the command-link for the command SPI2_WHOAMIcmd. It's important to notice that the command-link and the command share almost identical names; SPI2_WHOAMIcmd and SPI2_WHOAMI_cmd.

To post the command SPI2_WHOAMIcmd from a different Device, the command-link SPI2_WHOAMI_cmd is posted.

```
@; Post command-link
MOV_imm32 r0, SPI2_WHOAMI_cmd
MOV_imm32 r1,internalCmds
bl MoT_cmdPost
```

Listing 3: Example command-link: SPI2 WHOAMI, II

Listing 3 shows how a command-link is posted.

4 Tasks

A task is reminiscent of a command, except, each Device can have exactly *one* installed task. The flowchart shown in chapter 2 on Figure 1 shows the order in which each Device's task is executed, and is simply deciphered by following the solid black arrows from Device0.

```
@; cycle-counting version of a blinking LED task
.global userLED_ONtask  @; changes to the alternate userLED_OFFtask
    after every ULED_RELOAD task cycles
```

```
.thumb_func
userLED_ONtask: @;  arrive here with rDEVP pointing to this device's data
ldr r1,[rDEVP,#ULED_COUNT] @; update cycle count
subs r1,#1                 @; ..
str r1,[rDEVP,#ULED_COUNT] @; ..
cbnz r1,1f                 @;
@; here when countdown is done
ldr r1,[rDEVP,#ULED_RELOAD]    @; restart cycle countdown
str r1,[rDEVP,#ULED_COUNT]     @; ..
MOV_imm32 r0,userLED_OFFtask   @; MoT_taskUpdate() arguments are the execution address of another task
mov r1,#NULL
bl MoT_taskUpdate  @; MoT_taskUpdate(userLED_OFFtask,NULL) will install the LED 'OFF' task
bl userLED_OFF @; turn the Green LED off in preparation for next task cycle
1:  @; task for this device is done for now -- proceed to next device/task on list
ldr rDEVP,[rDEVP,#NEXTTASK]    @; get new rDEVP
ldr pc,[rDEVP,#XEQTASK]        @; start task of new device
```

Listing 4: Example task: userLED ON

Listing 4 shows the `userLED_ONtask` of the userLED Device. We'll use this task to explore the concept of how they work, what they do, and how to implement them.

The task, `userLED_ONtask`, is responsible for managing the `ON` state of a user LED in a cycle-counting manner. The task is designed to toggle the LED state after a specific number of task cycles, as determined by a countdown value stored in a variable at offset `ULED_COUNT`. The value stored in the variable is decremented by one and stored in the variable again. A check is then performed to see whether `0x00` has been reached or not; if it *hasn't*, the task branches forward to the label `1`, if it *has* a value from a different variable at offset `ULED_RELOAD` is stored into variable `ULED_COUNT`.

As the task has finished its job *and* `0x00` has been reached, a new task is installed.

```
MOV_imm32 r0,userLED_OFFtask   @; MoT_taskUpdate() arguments are the execution address of another task
mov r1,#NULL
bl MoT_taskUpdate  @; MoT_taskUpdate(userLED_OFFtask,NULL) will install the LED 'OFF' task
```

Listing 5: Example task: userLED ON

Listing 5 shows the installation of the task `userLED_OFFtask`. This is done by moving the address of the task into register `r0`. It's important to note that `NULL` must also be moved into `r1`, and is defined as `.equ NULL, 0` in each Device.

As the task address has been moved to `r0` and `r1` holds `NULL`, the function `MoT_taskUpdate` is branched to and `userLED_OFFtask` will be executed the next time the Device is reached; the task has been *installed*. To turn the LED off, the function `userLED_OFF` is then branched to. If however, `0x00` has not been reached and the task forward branched to label `1` as stated earlier, `userLED_ONtask` is still the installed task and will be executed next time the Device is reached.

To conclude the execution of the task, the next Device in line's task is executed with `ldr rDEVP,[rDEVP,#NEXTTASK]` and `ldr pc,[rDEVP,#XEQTASK]`.

Note: Each Device has one common task: `DeviceX_skiptask`. This task does nothing, but since each Device must have an installed task, this is necessary.

5 Devices

5.1 Variables & Buffers

A Device can use variables to store up to a 32-bit value, just like a `.word` is generally invoked in the `.data` section, and they can use buffers to store a set amount of bytes; a series of `.byte` in `.data`. The main difference between MoT variables and a general `.word`, is that an MoT variable is faster to access.

```
MoT_varAlloc_m SPI2_cmdargumentsaddress, SPI2_CMDARGSADDR, 0x00
```

Listing 6: Variable Declaration: SPI2 Command Argument Address

The declaration of an MoT variable is shown on Listing 6.[1] The macro `MoT_varAlloc_m` is invoked with arguments `SPI2_cmdargumentsaddress, SPI2_CMDARGSADDR, 0x00`.

`SPI2_cmdargumentsaddress` is the label for the absolute address of the variable, while `SPI2_CMDARGSADDR` represents the offset applied to `rDEVP` during load or store operations.

```
@; Store value in r0 in variable SPI2_cmdargumentsaddress using
   SPI2_CMDARGSADDR offset; 1 instruction
str r0, [rDEVP, #SPI2_CMDARGSADDR]

@; Store value in r0 in variable SPI2_cmdargumentsaddress using
   absolute address; 3 instructions (MOV_imm32 is a 2 instructions
   macro)
MOV_imm32 r1, SPI2_cmdargumentsaddress
str r0, [r1]
```

Listing 7: Variable Example

[1]The snippet is taken from the SPI2 Device.

Once a variable has been declared using `MoT_varAlloc_m`, it can be accessed in different ways depending on the context, as shown in Listing 7. The first example uses the offset of the variable applied to `rDEVP` which only requires *one* instruction. The second approach is to use the absolute address, which is required when accessing the variable from a different Device. However, this method requires at least *three* instructions:

`MOV_imm32 r1, SPI2_cmdargumentsaddress`: Two instructions macro.

`str r0, [r1]`: One instruction.

MoT variables are essentially faster to access because their offsets are predefined at compile time relative to `rDEVP`, eliminating the need for additional instructions to calculate or load their absolute address.

Since the G49 MoT processes Devices sequentially, minimizing instruction usage is crucial to reduce the time spent on each Device.

5.2 Messages

A message is simply a buffer that holds the ASCII characters to transmit back to the user via UART. In a user-less system, this could also be used to return information to a different system.

```
1 MoT_msgLink_m INITIALIZED_msg, INITIALIZED_msgtxt,
2 INITIALIZED_msglen,
3
4 INITIALIZED_msgtxt:
5 .ascii "SPI2 initialized!\n\r\0"
6 .equ INITIALIZED_msglen, ( . - INITIALIZED_msgtxt)
```

Listing 8: Constant Message Example: SPI2 Initialization

Listing 8 shows the `INITIALIZED` message for the SPI2 Device. This is done with the macro `MoT_msgLink_m` in ROM. `INITIALIZED_msg` is the label for the address used when posting the message, `INITIALIZED_msgtxt` is the label defined on line 4, which is the message itself. `INITIALIZED_msglen` is the length of the message. Do note, that a trailing comma follows at the end. This is *necessary*. For a constant message this is simply defined as the difference between the address of label `INITIALIZED_msgtxt` and the address where the length is calculated; see line 6.

The discussed message is *constant*. Once defined, that's what it is. However, it can be useful to adjust the message based on received data or calculations done in a command. Therefore, we also have *dynamic* messages. These are adjustable at runtime due to their reliance on buffers in RAM.

```
.equ RECEIVED_msglen, 0x15
```

```
MoT_printbuffer_m RECEIVED_msgbuf, RECEIVED_msglen
MoT_msgLink_m RECEIVED_msg, RECEIVED_msgbuf, RECEIVED_msglen,
```

Listing 9: Dynamic Message Example: SPI2 Received.

This functionality is achieved using the same macro `MoT_msgLink_m` seen on Listing 9, but in combination with `MoT_printbuffer_m`. This allows for a dynamic message, by linking to a buffer. `MoT_printbuffer_m` defines the buffer and reserves space in RAM of size `RECEIVED_msglen` bytes; `0x15`.

```
uint32_t device3_printMSGRX(char *buf, char data) {
    sprintf(buf,"SPI2 received 0x%02X!\r\n", data);
    return strlen(buf);
}
```

Listing 10: Example message: SPI2 Received

The function `device3_printMSGRX` in Listing 10 is created using mpaland's embedded printf library.[2] Such functions are defined in the *mpaland_printf_services.c* file. To use it, it can be branched to by a command or a task, which has the buffer address and the data to write placed in `r0` and `r1` respectively.

```
ldr r0, =RECEIVED_msgbuf     @; Load MoT message buffer
bl device3_printMSGRX        @; Populate buffer with received byte
```

Listing 11: Example usage of device3_printMSGRX

An example of branching to a similar function can be found multiple places in the SPI2 Device, but above on Listing 11 is included the branching to the function in Listing 10, with data received from the W25Q128 module in `r1`.[3] It's important to note that `=RECEIVED_msgbuf` is the address of the message buffer.

Messages are an important part of the system, and simultaneously they are very easy to implement and adjust. In addition, just as each Divce has a `skiptask`, they also have a `skip` message.

5.3 Structure

The structure of a Device is *fixed*. This means that certain operations must be done in the correct order; otherwise, the Device will fail, potentially leading to system-wide issues.

5.3.1 MoT_main.c

The very first thing that happens doesn't happen in the Device. It happens in `MoT_main.c`.

[2] `https://github.com/mpaland/printf`

[3] The snippet is from the SPI2 Device task `SPI2_INSTRUCTIONtask`.

The MoT Device must be defined in the main file as an `MoT_core_t`: `extern MoT_core_t Device1;`. In this case *`Device1`* is a placeholder name, but the name isn't irrelevant; it *must* match the name of the MoT Device defined in the respective MoT file.

For a Device to be included in the system, it must first be inserted into the list of Devices. This list lives in `MoT_main.c` and is defined as `MoT_core_t *devicelist[] = {&device0, &Device1, &Device2, ..., &deviceN}`. The position at which a Device is placed in this list determines the Device number used in the command string. As `Device1` is second in the list, - but we count from *zero* - it's Device number is `1`. As explained, the name is connected to the Device's MoT file; `Device1` and `Device2` are simply placeholder names. It's **very** important to note that `Device0` *must* be first and `DeviceN` *must* be last; this is *fixed*.

As the Device has been defined in `MoT_main.c`, the MoT Device file is included in the system.

5.3.2 MoTdevice_DeviceX.S

When setting up the MoT Device the order of operations is crucial.

```
#define __ASSEMBLY__    @;for MoTstructures.h -- selects assembly
    macro definitions instead of C structs
#include "MoTstructures.h"
.equ NULL,0    @;for returning NULL pointr various places

@;register equates
rLISTP   .req r4 @;points to the device_list[] defined in main()
rDEVP    .req r5 @;points to the device data of the current active
    task.
```

Listing 12: MoT Device file structure, I.

As seen in Listing 12, the very first thing that's done is `#define __ASSEMBLY__`. This enables the use of macros, such as the `MoT_varAlloc_m`, as it would otherwise be a C macro; rendering it unusable in assembly. These macros are defined in `MoTstructures.h`, which is included immediately after. As seen earlier, the use of `NULL` happens quite often. `NULL` is here defined as a replacement for `0`.

When accessing, say, variables, using an offset, that offset is applied to the `rDEVP` register. This is defined as the equivalent for register `r5`; if this step is excluded the Device will fail.

For the system to access the next Device in the list, the equivalent for register `r4` is defined as `rLISTP`. This register is not used for data access, but to reach the following Device.

As the above steps are completed, the construction of the Device begins.

```
    .syntax unified @; ARM Unified Assembler Language (UAL) is
        allowed
    .thumb @; we are using thumb instructions only

    .text   @; macros below place variable data in initialized .data,
       the rest in .text

    .global DeviceX_cmdHandler
    .thumb_func
DeviceX_cmdHandler:
    push {r7,lr}
    ldrb r1,[r0],#1     @; read command from the command buffer and
       advance r0 to point to the argument list (if any)
    and r1,0x07     @; limit range of command codes to 0-7 for safety

    tbb [pc,r1]
DeviceX_cmds:
    .byte (DeviceX_INITcmd - DeviceX_cmds)/2
    .byte (DeviceX_YYcmd - DeviceX_cmds)/2
    .byte (DeviceX_SKIPinstall - DeviceX_cmds)/2
    .align 1

    MoT_core_m DeviceX, DeviceX_cmdHandler, DeviceX_skiptask
```

Listing 13: MoT Device file structure, II.

After setting up the assembly macros and register definitions, the next step involves defining the Device's command handler and associated table, as shown in Listing 13. The concept behind the command handler and table is straightforward. Each Device contains a `cmdHandler`, in this case `DeviceX_cmdHandler`, which branches to the appropriate command in `cmds` given in the command string as dispatched from `Device0`. To navigate commands, the system uses a `byte` table, although a `half-word` table can also be implemented just by ajusting the entries in `cmds` to be `.hword` and using `tbh [pc,r1]`. Only side effect is, that this will take up more memory; currently not an issue for the G49 MoT system.

The `cmdHandler` serves as the entry point for processing commands dispatched to this Device, while the `cmds` table maps command codes to their respective labels.

Here it should be noted that the macro `MoT_core_m` defines the Device. In this case it's DeviceX, but using the example from chapter 5.3.1 `MoT_main.c` we would write `MoT_core_m Device1, Device1_cmdHandler, Device1_skiptask`. The last arguement in the macro is `DeviceX_skiptask`, this is the default task that is installed. It can essentially be any task in the Device, however, `skiptask` is used in every Device already implemented.

The above steps ensure that each Device is properly configured to handle commands within the G49 MoT system; duplicating an existing Device and adjusting as needed is generally an efficient way of creating a new Device.

6 DAC1

The DAC1 Device is used to output an analog signal to one of the GPIO pins on the STM32G491RE development board. Without any adjusting, this signal is output to the userLED, PA5, pin. This however, could be changed without much trouble, depending on the availability of pins on the different DAC1 channels. Read the manual for more information.

6.1 Device Variables

The declaration of variables used by the Device is shown in listing 14 in the appendix. The purpose of each is explained in the table below:

Variable	**Description**	**Commands**	**Tasks**
DAC1_T1	Stores the width of the first part of a requested pulse.	DAC1_PULSEcmd	None
DAC1_T2	Stores the width of the second part of a requested pulse.	DAC1_PULSEcmd	DAC1_pulsetaskT1
DAC1_LOW	Stores the voltage of the first part of an executed pulse.	DAC1_PULSEcmd	None
DAC1_HIGH	Stores the voltage of the second part of a requested pulse.	DAC1_PULSEcmd	DAC1_pulsetaskT1
DAC1_SIN_EN	Used as an indicator for whether or not a sine wave is being produced.	DAC1_SINcmd	None
DAC1_SIN_POS	Stores the position of the sine wave value to output.	None	None
DAC1_SIN_AMPL	Stores the amplitude of the sine wave output.	DAC1_SINcmd	None

Table 2: DAC1 MoT Device Variables

Table 2 shows every variable used by the DAC1 MoT Device, where it's used and what it's used for. Notice that DAC1_SIN_POS is not used in this Device directly, it is however used in the interrupt service rutine for TIM4, which is used to control the frequency of the sine wave.

6.2 Device Messages

Declaration of messages used in the DAC1 Device is shown in the appendix on Listing 15.

Message	Commands	Tasks
INITIALIZED	DAC1_INITcmd	None
CONSTANT	DAC1_CONSTANTcmd	None
PULSE	DAC1_PULSEcmd	None
SIN	DAC1_SINcmd	None
SIN_STOP	DAC1_SINcmd	None

Table 3: DAC1 MoT Device Variables

The messages shown in Table 3 are the ones used to communicate with the user, letting them know that the command has been executed. The contents of the messages can also be seen in the appendix on Listing 15.

6.3 Device Commands

This section contains a table of the DAC1 Device commands that are currently implemented. Following the table is a summary of each command, and in the appendix is located a flowchart for each command. For a deeper understanding of each command, they should be individually examined in the `MoTdevice_DAC1.S` file.

Command	Usage	Inputs
DAC1_INITcmd	Initializes the DAC1 MoT Device.	None
DAC1_CONSTANTcmd	Outputs a constant voltage to the userLED pin.	12-bit
DAC1_PULSEcmd	Outputs a pulse using TIM2.	16-bit, 16-bit, 12-bit, 12-bit
DAC1_SINcmd	Outputs a sine wave using TIM4.	16-bit, 12-bit

Table 4: DAC1 MoT Device Commands

The commands for the DAC1 MoT Device is shown on Table 4.

`DAC1_INITcmd`: The command initializes PA5 and configures it to analog mode, and `DAC1_CH2` is enabled. This is done by branching to the LL functions `DAC1_CH2_init` and `GPIOA_init`. To conclude the command, the message `INITIALIZED` is posted. The command string for

the command is as follows:
`:0100FF`

`DAC1_CONSTANTcmd`: The command outputs a given voltage level on pin PA5. This is done by loading the input and branching to the LL function `DAC1_CH2_constant_output`. The voltage level is constrained to a 12-bit value. The command posts the message `CONSTANT` at the end. An example of a command string could be:
`:01015505A4`

`DAC1_PULSEcmd`: The command outputs a single pulse on pin PA5 using TIM2. The pulse is defined by four inputs; 16-bit time delay, 16-bit time delay, 12-bit voltage level, 12-bit voltage level. The time delay inputs are converted to μs to work with the `TIM2_pulse_usec`. The voltage levels are sorted so the lowest level is first and the largest second. The commands relies on the two tasks `DAC1_pulsetask1` and `DAC1_pulsetask2`. An example of a command string could be:
`:0102C409C409FF0F5505FB`

`DAC1_SINcmd`: The command either starts or stops the output of a sine wave. The command takes two inputs; 16-bit frequency, 12-bit amplitude. If a sine wave is currently being output, it's paused, otherwise it's started by initializing TIM4. TIM4 is given an ARR value calculated from the given frequency using the formula:

$$ARR = \frac{36MHz}{freq * 128} - 1$$

The $36MHz$ is the TIM4 interrupt frequency and 128 is the number of values the sine wave LUT consists of. An example of a command string could be:
`:01036400FF0F8A`

That concludes the commands of the DAC1 Device.

6.4 Device Tasks

This section contains a table of the DAC1 Device tasks that are currently implemented. Following the table is a summary of each task, and in the appendix is located a flowchart for each task. For a deeper understanding of each task, they should be individually examined in the `MoTdevice_DAC1.S` file.

Task	**Usage**	**Inputs**
DAC1_pulsetaskT1	Poll first pulse; update new pulse width and voltage level when done.	DAC1_T2, DAC1_HIGH
DAC1_pulsetaskT2	Poll second pulse; clear voltage level when done.	None

Table 5: DAC1 MoT Device Tasks

The tasks for the DAC1 MoT Device is shown on Table 5.

`DAC1_pulsetaskT1`: The task polls for a running pulse by polling `TIM2_CEN`. If no pulse is running, that means the pulse generated by `DAC1_PULSEcmd` is done and the task stores the high voltage level in the DAC output holding buffer, and starts a new pulse with width `DAC1_T2`.

`DAC1_pulsetaskT2`: The task polls for a running pulse by polling `TIM2_CEN`. If no pulse is running, that means the pulse generated by `DAC1_pulsetaskT1` is done and the task sets the voltage level to $0V$ and installs the skip task.

7 TIM2

The TIM2 Device is used to output a single timed output, or a stream of scheduled outputs, to one of the GPIO pins on the STM32G491RE development board. Without any adjusting, this signal is output to the userLED, PA5, pin. This however, could be changed without much trouble, depending on the availability of pins connected to the TIM2 channels. Read the manual for more information.

7.1 Device Variables

The declaration of variables used by the Device is shown in listing 16 in the appendix. The purpose of each is explained in the table below:

Variable	**Description**	**Commands**	**Tasks**
TIM2_DELAY	Stores the delay until a requested pulse starts.	*	*
TIM2_DURATION	Stores the width of an ongoing pulse.	*	*

Table 6: TIM2 MoT Device Variables

Table 6 shows every variable used by the TIM2 MoT Device.
**All variables are used by the TIM2_STREAMcmd and TIM2_STREAMtask.*

7.2 Device Messages

Declaration of messages used in the TIM2 Device is shown in the appendix on Listing 17.

Message	**Commands**	**Tasks**
INITIALIZED	TIM2_INITcmd	None
PULSE_ONE	TIM2_PULSE_ONEcmd	None
PULSE_STREAM	TIM2_PULSE_STREAMcmd	None

Table 7: TIM2 MoT Device Variables

The messages shown in Table 7 are the ones used to communicate with the user, letting them know that the command has been executed. The contents of the messages can also be seen in the appendix on Listing 17.

7.3 Device Commands

This section contains a table of the TIM2 Device commands that are currently implemented. Following the table is a summary of each command, and in the appendix is located a flowchart for each command. For a deeper understanding of each command, they should be individually examined in the `MoTdevice_TIM2.S` file.

Command	**Usage**	**Inputs**
TIM2_INITcmd	Initializes the TIM2 MoT Device.	None
TIM2_PULSE_ONEcmd	Outputs a single pulse using on PA5.	16-bit, 16-bit
TIM2_PULSE_STREAMcmd	Outputs a stream of pulses on PA5.	16-bit, 16-bit

Table 8: TIM2 MoT Device Commands

The commands for the TIM2 MoT Device is shown on Table 12.

`TIM2_INITcmd`: The command initializes PA5, and `TIM2_CH2` is enabled. This is done by branching to the LL functions `TIM2_init` and `TIM2_connect_CH1_to_PA5`. At the end, the command posts the message `INITIALIZED`. The command string for the command is as follows:
`:0200FE`

`TIM2_PULSE_ONEcmd`: The command outputs a single pulse on pin PA5, given two inputs: a delay and a duration, both in msec. Both inputs are limited to 16-bit. The command converts the inputs to μs and branches to the LL function `TIM2_pulse_usec`. The command posts the message `PULSE_ONE` at the end. An example of a command string could be:
`:0201D007D0074F`

`TIM2_PULSE_STREAMcmd`: The command outputs a stream of pulses on pin PA5, given two inputs: a delay and a duration, both in ms. Both inputs are limited to 16-bit. The command converts the inputs to μs and stores them in the variables `TIM2_DELAY` and `TIM2_DURATION`. The command then branches to the LL function `TIM2_pulse_usec` and installs the task `TIM2_PULSE_STREAMtask`. The command posts the message `PULSE_STREAM` at the end. An example of a command string could be:
`:02026400D007C1`

That concludes the commands of the TIM2 Device.

7.4 Device Tasks

This section contains a table of the TIM2 Device tasks that are currently implemented. Following the table is a summary of each task. For a deeper understanding of each task, they should be individually examined in the `MoTdevice_TIM2.S` file.

Task	**Usage**	**Inputs**
TIM2_PULSE_STREAMtask	Polls for a running pulse; if none detected then start a new using the input variables and branching to the LL function `TIM2_pulse_usec`.	TIM2_DELAY, TIM2_DURATION

Table 9: TIM2 MoT Device Tasks

The tasks for the TIM2 MoT Device is shown on Table 9.

`TIM2_PULSE_STREAMtask`: The task polls `TIM2_CR1_CEN` for a running pulse. If a pulse is currently being produced, the task branches to the exit. However, if no pulse is detected (the previous pulse is finished), the task will load the variables `TIM2_DELAY` and `TIM2_DURATION` and branch to the LL function `TIM2_pulse_usec`.

8 SPI2

The SPI2 Device is used to communicate with an external module requiring the SPI communications protocol. It is *not* configured to work with more than one (1) slave. Currently, the Device has the ability to do a WHOAMI check, for which we expect the Manufacture ID 0xEF in response, it can read and write a byte at a 24-bit address, it can erase a 64 kB block, read a byte at a 24-bit address, and write up to 256 bytes at a 24-bit address. The Device has been developed with modularity in mind, making it straightforward[4] to add functionality; additional instructions, different module support, etc...

[4]The current commands should be analyzed to understand the setup necessary for a command. Additionally, Fig 10, 11, 12, 13, and 14 should be referenced.

8.1 Device Variables

The declaration of variables used by the Device is shown in listing 18 in the appendix. The purpose of each is explained in the table below:

Variable	**Description**	**Commands**	**Tasks**
SPI2_CMDARGS-ADDR	Stores the address of the Device arguments.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	None
SPI2_PRINT-RXBYTE	Indicates whether the SPI2_INSTRUCTION-task should print the received byte or not.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	S..._IN..ONtask
SPI2_RXBYTE	Stores the byte received from the external flash module.	None	S..._IN..ONtask
SPI2_FAILS	Stores the number of failed receive/transmit attempts.	None	S..._IN..ONtask
SPI2_INSTRUCTI-ONSTEP	Stores the next step of the INSTRUCTION-task.	None	S..._IN..ONtask
SPI2_INSTRUCTI-ONLENGTH	Stores the length of the current instruction to transmit.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	S..._IN..ONtask
SPI2_CMD	Stores the command to be branched back to from the INSTRUC-TIONtask.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	S..._IN..ONtask
SPI2_CMDSTEP	Stores the next step of the current command.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	None
SPI2_TXBUFFER	Stores all the bytes to transmit.	S..._INITcmd, S..._ER...64cmd, S..._RE...TEcmd, S..._WR...TEcmd	S..._IN..ONtask

Table 10: TIM2 MoT Device Variables

Table 6 shows every variable used by the TIM2 MoT Device. The command and task names

have been shortened to fit the table, these are the full names:
`S..._INITcmd` → `SPI2_INITcmd`
`S..._ER...64cmd` → `SPI2_ERASEBLOCK64cmd`
`S..._RE...TEcmd` → `SPI2_READBYTEcmd`
`S..._WR...TEcmd` → `SPI2_WRITEBYTEcmd`
`S..._IN...ONtask` → `SPI2_INSTRUCTIONtask`

8.2 Device Messages

Declaration of messages used in the TIM2 Device is shown in the appendix on Listing 19.

Message	Commands	Tasks
INITIALIZED	SPI2_INITcmd	None
WHOAMI	SPI2_WHOAMIcmd	None
ERASE64	SPI2_ERASEBLOCK64cmd	None
READ	SPI2_READBYTEcmd	None
WRITING	SPI2_WRITEBYTEcmd	None
DONE	SPI2_SLAVEBSYcmd	None
ERROR	None	SPI2_INSTRUCTIONtask
RECEIVED	None	SPI2_INSTRUCTIONtask

Table 11: SPI2 MoT Device Variables

The messages shown in Table 11 are the ones used to communicate with the user, letting them know that the command has been executed. The contents of the messages can also be seen in the appendix on Listing 19. The `RECEIVED` message is a dynamic message.

8.3 Device Commands

This section contains a table of the SPI2 Device commands that are currently implemented. Following the table is a summary of each command, and in the appendix is located a flowchart for each command. For a deeper understanding of each command, they should be individually examined in the `MoTdevice_SPI2.S` file.

Command	**Usage**	**Inputs**
SPI2_INITcmd	Initializes the SPI2 MoT Device.	None
SPI2_WHOAMIcmd	Reads the Manufacture ID from the external flash module.	None
SPI2_ERASEBLOCK64cmd	Erases a block of memory at a given 24-bit address.	24-bit (split into 3 $\times$ 8-bit)
SPI2_READBYTEcmd	Reads a byte at a given 24-bit address.	24-bit (split into 3 $\times$ 8-bit)
SPI2_WRITEBYTEcmd	Writes between 1 and 256 bytes of data to a given 24-bit address. If more than one byte is given as input data, the bytes propagate the next address until done.	8-bit, 24-bit, 8-bit to 2048-bit (split into 8-bit).
SPI2_SLAVEBSYcmd	Polls the BSY bit of the external flash module.	None

Table 12: TIM2 MoT Device Commands

The commands for the SPI2 MoT Device is shown on Table 12.

`SPI2_INITcmd`: The command initializes the SPI2 peripheral and performs the following configuration: Master, Slave Select Output, 8-bit data size, 1/4 FIFO Threshold, Baud Rate Pre-scaler of 16, Slave Software Management. Additionally GPIOB pin PB12, PB13, PB14, and PB15 is configured to work as CS, SCK, MISO, and MOSI respectively. This means setting PB13, PB14, and PB15 in alternate function.

`SPI2_WHOAMIcmd`: Sends the JEDEC ID instruction to the external flash module. This in turn makes the `SPI2_INSTRUCTIONtask` post the message `RECEIVED` which should be populated with the Manufacture ID, `0xEF`.

`SPI2_ERASEBLOCK64cmd`: The command erases a block of memory at a given 24-bit address by sending the instruction Block Erase (64KB) to the external flash module. However, this instruction requires that the module is put into write-mode by first sending the instruction Write Enable. As the instruction Block Erase (64KB) has been transmitted the command posts the inter-Device command `SPI2_SLAVEBSY_cmd`.

`SPI2_READBYTEcmd`: The command reads one byte at a 24-bit address. This is done by sending the instruction Read Data to the external flash module. The task `SPI2_INSTRUCTIONtask` prints the received byte.

`SPI2_WRITEBYTEcmd`: The command writes up to 256 bytes of data to a 24-bit address in the external flash module by sending the instruction Page Program. However, just like the command SPI2_ERASEBLOCK64cmd, the Write Enable instruction must be send first. As the instruction Page Program has been transmitted the command posts the inter-Device command SPI2_SLAVEBSY_cmd.

SPI2_SLAVEBSYcmd: This command is an inter-Device command, with command-link label SPI2_SLAVEBSY_cmd. It can be used by the user, but it's generally useless for them. For inter-Device use, it's used after a write-operation. For the built-in commands this includes SPI2_ERASEBLOCK64cmd and SPI2_WRITEENABLEcmd. The command sends the instruction Read Status Register 1 to the external flash module and polls for the BSY bit.

8.4 Device Tasks

This section contains a table of the SPI2 Device tasks that are currently implemented. Following the table is a summary of each task. For a deeper understanding of each task, they should be individually examined in the MoTdevice_SPI2.S file.

Task	**Usage**	**Inputs**
SPI2_INSTRUCTIONtask	Transmit the SPI2_TXBUFFER content to the external flash module.	SPI2_PRINTRXBYTE, SPI2_RXBYTE, SPI2_FAILS, SPI2_INSTRUCTIONSTEP, SPI2_INSTRUCTIONLENGTH, SPI2_CMD, SPI2_TXBUFFER

Table 13: SPI2 MoT Device Tasks

The tasks for the SPI2 MoT Device is shown on Table 13.

SPI2_INSTRUCTIONtask: This task is the systems most complex task. It consists of multiple different steps, which in the right order transmits an instruction to the external flash module, prints the received byte (if necessary) and branchess back to the command that sent the instruction; if however the command that sent the instruction has specified so, the task branches to the SPI2_SLAVEBSYcmd using the command-link SPI2_SLAVEBSY_cmd.

9 Appendix

9.1 DAC1 Variables

```
MoT_varAlloc_m DAC1_delay1 , DAC1_T1
MoT_varAlloc_m DAC1_delay2 , DAC1_T2
MoT_varAlloc_m DAC1_Vlo , DAC1_LOW
MoT_varAlloc_m DAC1_Vhi , DAC1_HIGH
MoT_varAlloc_m DAC1_sin_enabled , DAC1_SIN_EN , 0x00
MoT_varAlloc_m DAC1_sin_position , DAC1_SIN_POS , 0x00
MoT_varAlloc_m DAC1_sin_amplitude , DAC1_SIN_AMPL , 0x0FFF
```

Listing 14: Variables for the DAC1 MoT Device

9.2 DAC1 Messages

```
MoT_msgLink_m INITIALIZED_msg, INITIALIZED_msgtxt, INITIALIZED_msglen
    ,
INITIALIZED_msgtxt:
.ascii "DAC1 is initialized\n\r\0"
.equ INITIALIZED_msglen, ( . - INITIALIZED_msgtxt)

MoT_msgLink_m CONSTANT_msg, CONSTANT_msgtxt, CONSTANT_msglen,
CONSTANT_msgtxt:
.ascii "DAC1 is outputting a constant voltage\n\r\0"
.equ CONSTANT_msglen, ( . - CONSTANT_msgtxt)

MoT_msgLink_m PULSE_msg, PULSE_msgtxt, PULSE_msglen,
PULSE_msgtxt:
.ascii "DAC1 is outputting a pulse\n\r\0"
.equ PULSE_msglen, ( . - PULSE_msgtxt)

MoT_msgLink_m SIN_msg, SIN_msgtxt, SIN_msglen,
SIN_msgtxt:
.ascii "DAC1 is outputting a sine wave\n\r\0"
.equ SIN_msglen, ( . - SIN_msgtxt)

MoT_msgLink_m SIN_STOP_msg, SIN_STOP_msgtxt, SIN_STOP_msglen,
SIN_STOP_msgtxt:
.ascii "DAC1 has stopped active sine wave\n\r\0"
.equ SIN_STOP_msglen, ( . - SIN_STOP_msgtxt)
```

Listing 15: Messages for the DAC1 MoT Device

9.3 DAC1 Commands

9.3.1 DAC1_INITcmd

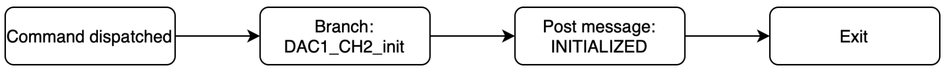

Figure 2: Flowchart of command DAC1_INITcmd

9.3.2 DAC1_CONSTANTcmd

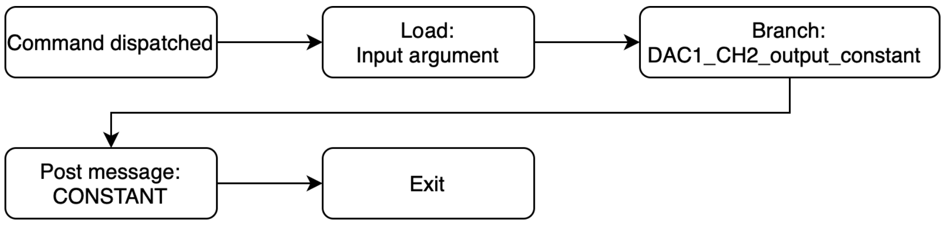

Figure 3: Flowchart of command `DAC1_CONSTANTcmd`

9.3.3 DAC1_PULSEcmd

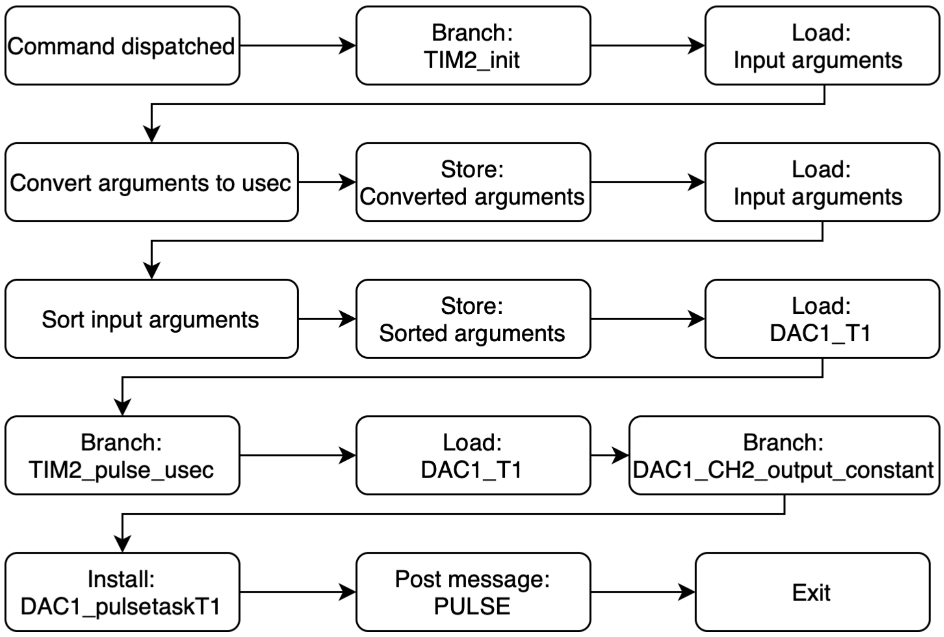

Figure 4: Flowchart of command `DAC1_PULSEcmd`

9.3.4 DAC1_SINcmd

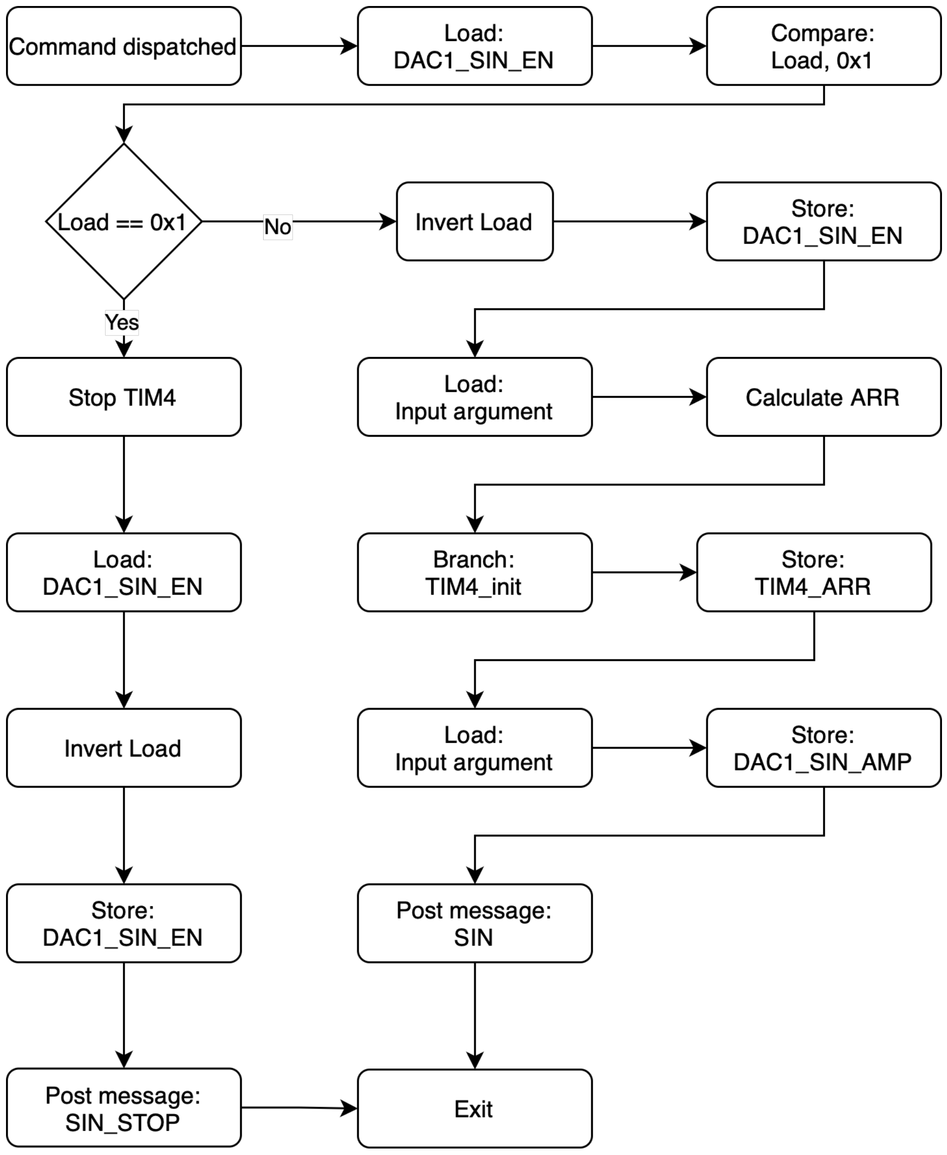

Figure 5: Flowchart of command `DAC1_SINcmd`

9.4 TIM2 Variables

```
MoT_varAlloc_m TIM2_delay, TIM2_DELAY
MoT_varAlloc_m TIM2_duration, TIM2_DURATION
```

Listing 16: Variables for the TIM2 MoT Device

9.5 TIM2 Messages

```
MoT_msgLink_m INITIALIZED_msg, INITIALIZED_msgtxt, INITIALIZED_msglen
    ,
INITIALIZED_msgtxt:
.ascii "TIM2 is initialized\n\r\0"
.equ INITIALIZED_msglen, ( . - INITIALIZED_msgtxt)

MoT_msgLink_m PULSE_ONE_msg, PULSE_ONE_msgtxt, PULSE_ONE_msglen,
PULSE_ONE_msgtxt:
.ascii "TIM2 is sending 1 (one) pulse\n\r\0"
.equ PULSE_ONE_msglen, ( . - PULSE_ONE_msgtxt)

MoT_msgLink_m PULSE_STREAM_msg, PULSE_STREAM_msgtxt,
   PULSE_STREAM_msglen,
PULSE_STREAM_msgtxt:
.ascii "TIM2 is generating pulses\n\r\0"
.equ PULSE_STREAM_msglen, ( . - PULSE_STREAM_msgtxt)
```

Listing 17: Messages for the TIM2 MoT Device

9.6 TIM2 Commands

9.6.1 TIM2_INITcmd

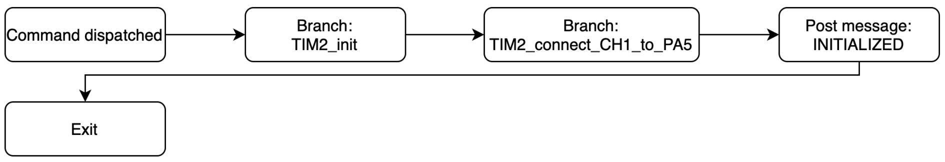

Figure 6: Flowchart of command TIM2_INITcmd

9.6.2 TIM2_PULSE_ONEcmd

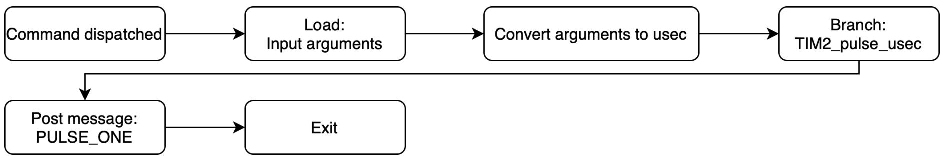

Figure 7: Flowchart of command TIM2_PULSE_ONEcmd

9.6.3 TIM2_PULSE_STREAMcmd

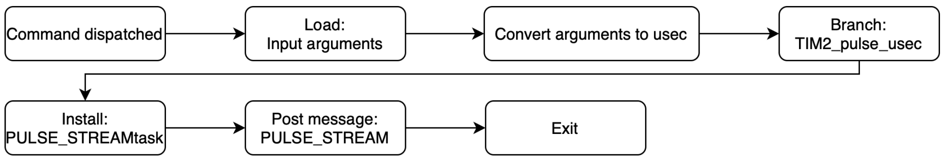
Command dispatched
Load: Input arguments
Convert arguments to usec
Branch: TIM2_pulse_usec
Install: PULSE_STREAMtask
Post message: PULSE_STREAM
Exit

Figure 8: Flowchart of the command TIM2_PULSE_STREAMcmd

9.7 SPI2 Variables

```
MoT_varAlloc_m SPI2_cmdargumentsaddress, SPI2_CMDARGSADDR, 0x00
MoT_varAlloc_m SPI2_printreceivedbyte, SPI2_PRINTRXBYTE, 0x00
MoT_varAlloc_m SPI2_receivebyte, SPI2_RXBYTE, 0x00
MoT_varAlloc_m SPI2_fail, SPI2_FAILS, 0x00
MoT_varAlloc_m SPI2_instructionstep, SPI2_INSTRUCTIONSTEP, 0x00
MoT_varAlloc_m SPI2_instructionlength, SPI2_INSTRUCTIONLENGTH, 0x00
MoT_varAlloc_m SPI2_command, SPI2_CMD, 0x00
MoT_varAlloc_m SPI2_commandstep, SPI2_CMDSTEP, 0x01
MoT_bufAlloc_m SPI2_transmitbuffer, SPI2_TXBUFFER, 0x2, 260
```

Listing 18: Variables for the SPI2 MoT Device

9.8 SPI2 Messages

```
.equ RECEIVED_msglen, 0x15
MoT_printbuffer_m RECEIVED_msgbuf, RECEIVED_msglen

MoT_msgLink_m RECEIVED_msg, RECEIVED_msgbuf, RECEIVED_msglen,

MoT_msgLink_m INITIALIZED_msg, INITIALIZED_msgtxt, INITIALIZED_msglen
    ,
INITIALIZED_msgtxt:
.ascii "SPI2 initialized!\n\r\0"
.equ INITIALIZED_msglen, ( . - INITIALIZED_msgtxt)

```

```
MoT_msgLink_m WHOAMI_msg, WHOAMI_msgtxt, WHOAMI_msglen,
WHOAMI_msgtxt:
.ascii "SPI2 is asking slave WHOAMI?\n\r\0"
.equ WHOAMI_msglen, ( . - WHOAMI_msgtxt)

MoT_msgLink_m ERASE64_msg, ERASE64_msgtxt, ERASE64_msglen,
ERASE64_msgtxt:
.ascii "SPI2 is erasing 64 KB block. Wait.\n\r\0"
.equ ERASE64_msglen, ( . - ERASE64_msgtxt)

MoT_msgLink_m CHIPERASE_msg, CHIPERASE_msgtxt, CHIPERASE_msglen,
CHIPERASE_msgtxt:
.ascii "SPI2 is erasing the chip. Wait.\n\r\0"
.equ CHIPERASE_msglen, ( . - CHIPERASE_msgtxt)

MoT_msgLink_m READ_msg, READ_msgtxt, READ_msglen,
READ_msgtxt:
.ascii "SPI2 is reading byte.\n\r\0"
.equ READ_msglen, ( . - READ_msgtxt)

MoT_msgLink_m WRITING_msg, WRITING_msgtxt, WRITING_msglen,
WRITING_msgtxt:
.ascii "SPI2 is writing byte.\n\r\0"
.equ WRITING_msglen, ( . - WRITING_msgtxt)

MoT_msgLink_m DONE_msg, DONE_msgtxt, DONE_msglen,
DONE_msgtxt:
.ascii "Done.\n\r\0"
.equ DONE_msglen, ( . - DONE_msgtxt)

MoT_msgLink_m ERROR_msg, ERROR_msgtxt, ERROR_msglen,
ERROR_msgtxt:
.ascii "SPI2 instruction failed!!\n\r\0"
.equ ERROR_msglen, ( . - ERROR_msgtxt)
```

Listing 19: Messages for the SPI2 MoT Device

9.9 SPI2 Commands

9.9.1 SPI2_INITcmd

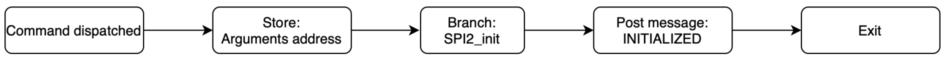

Figure 9: Flowchart of SPI2_INITcmd

9.9.2 SPI2_WHOAMIcmd

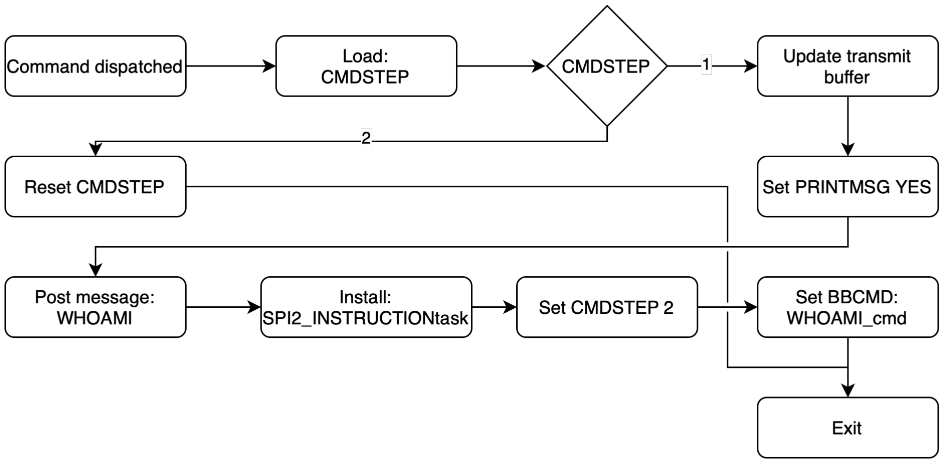

Figure 10: Flowchart of SPI2_WHOAMIcmd

9.9.3 SPI2_ERASEBLOCK64cmd

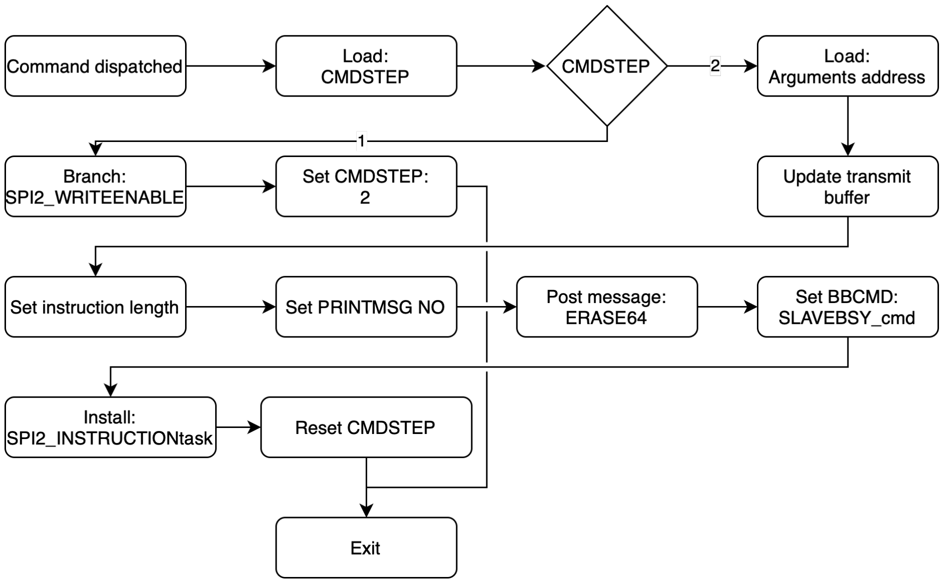

Figure 11: Flowchart of SPI2_ERASEBLOCK64cmd

9.9.4 SPI2_READBYTEcmd

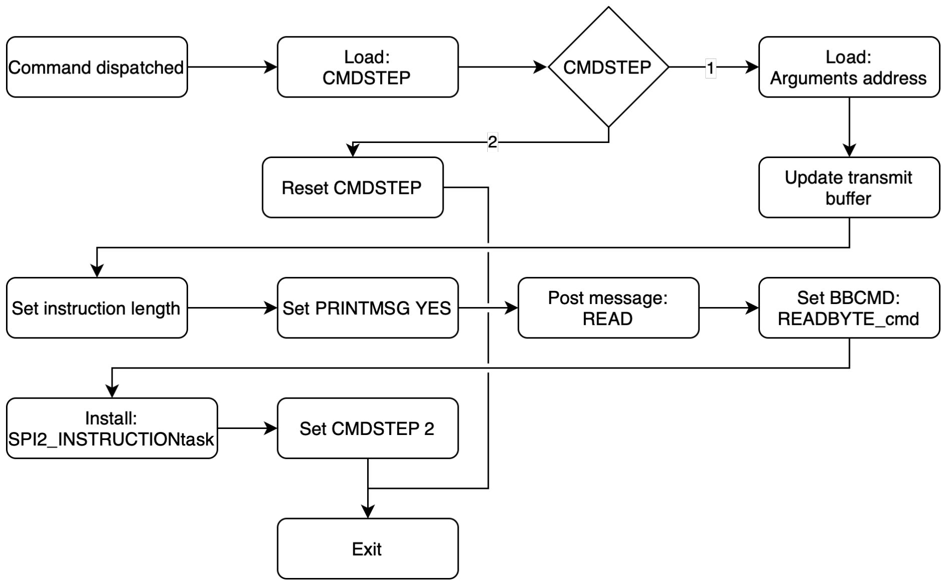

Figure 12: Flowchart of SPI2_READBYTEcmd

9.9.5 SPI2_WRITEBYTEcmd

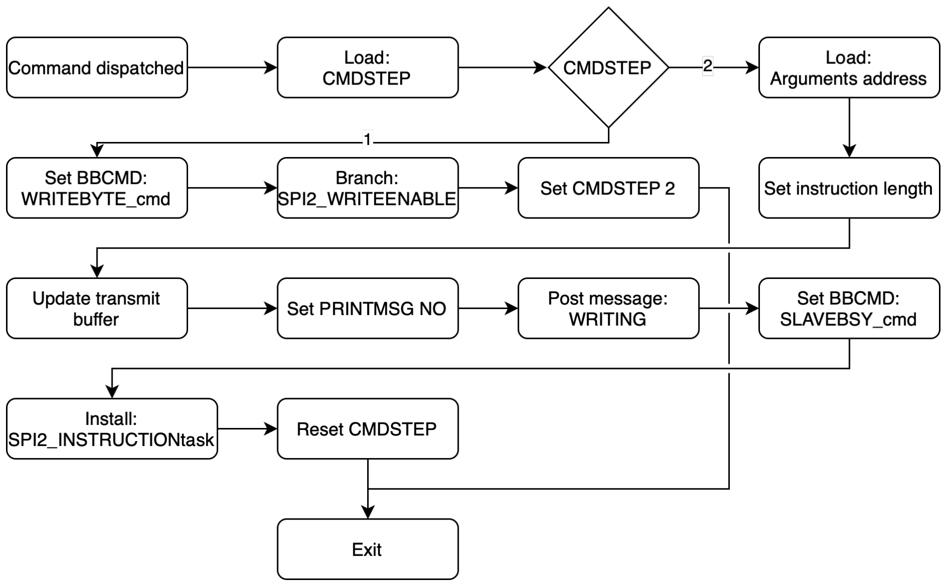

Figure 13: Flowchart of SPI2_WRITEBYTEcmd

9.9.6 SPI2_SLAVEBSYcmd

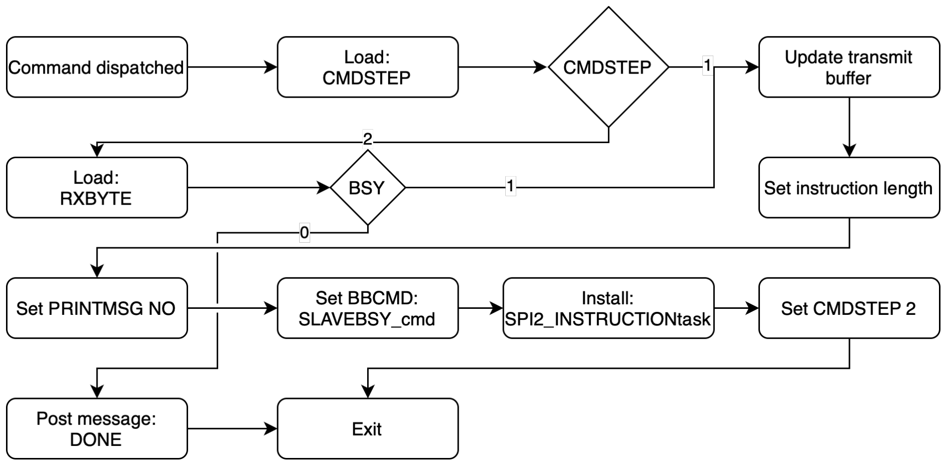

Figure 14: Flowchart of SPI2_SLAVEBSYcmd