DC11544
Driver 2.21
July 2023
R. Ertolupi

Interfacing FlashRunner with Power PC MCUs (MPC57xx, SPC58xx and S32R2xx)

1. Introduction

As you may know, Power PC MCUs are not very new on the market, they are based on the Power PC architecture which was developed by Apple, IBM, and Motorola in the early '90s. Nowadays ST Microelectronics and NXP (Freescale) are the ones producing these powerful microcontrollers and they are very common for automotive applications, especially for safety tasks, such as:

- Airbag management.
- Electronic stability control (ESC).
- Anti-lock braking system (ABS).
- Higher-end Electrical Power Steering (EPS).
- Battery management.
- Engine management.
- RADAR applications for advanced driver-assistance systems (ADAS).

If you are familiar with this kind of devices, you should know how many peculiarities they have and how complex they are. In this document, we are also explaining how FlashRunner can properly program these Power PC MCUs and manage their special features.

You can download the latest version of this document from this static link: [Interfacing FlashRunner with Power PC MCUs]().

2. Contents

3. Tips and tricks to flash Power PC MCUs

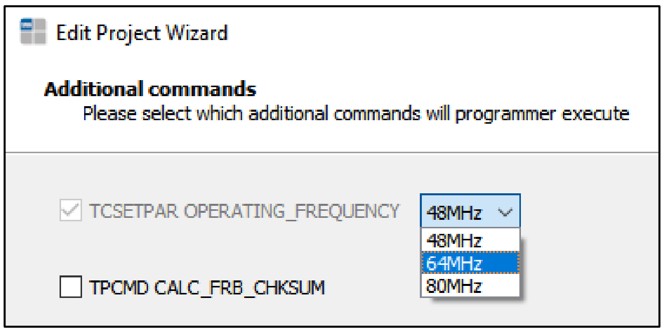
In this section, we want to explain what operations are performed in the target devices and how these operations work. This knowledge may help you to optimize even more your application.

The FlashRunner uses JTAG to access to the memories of these devices, which is one of the most common protocols for microcontrollers and which needs four synchronous lines (TCK, TMS, TDI, and TDO) and an optional asynchronous line (TRST). In addition to standard JTAG lines, the FlashRunner also need to control the reset line of the target device: this line is very important and very critical as well. It is crucial to have no interferences on the reset line because it could cause strange behaviours of the device and the operations could likely fail.

The new driver, combined with an OS > 3.00, can support a clock frequency for JTAG up to 37.5 MHz, which also helps to reduce flashing times. However, there are a couple of things to say about that.
The first one is that the customer needs a good hardware setup to be able to reach such a high frequency, otherwise the quality of the signals will be bad and operations could fail.
The second aspect to consider is that the JTAG frequency is proportionally limited to the system clock frequency (typically the limit is SYS_CLK /2 or /4). Exactly for this reason it is fundamental to enable the PLL by setting the following parameter to YES.

```
#TCSETPAR PLL_EN YES
```

This parameter is set to YES by default and it will configure the target device to work extremely near the maximum frequency. For some devices, especially the ones which are less powerful, even by enabling the PLL, they cannot support 37.5 MHz but the user should not worry about that because the driver is checking if the required specifications are respected and it automatically will lower the JTAG frequency if needed.
Please, note that the configuration of the PLL done by the FlashRunner is volatile and it will be lost after resetting the device, so it is completely transparent to the customer.
Some devices are produced in various versions which may significantly differ for the operating frequency. To exploit the best performances from the device, we offer to the user the possibility to explicitly choose which operating frequency to use according to the part number of the target device.
This is the example for SPC582Bxx which has the following three options:


Edit Project Wizard

Additional commands
Please select which additional commands will programmer execute

☑ TCSETPAR OPERATING_FREQUENCY [48MHz]
☐ TPCMD CALC_FRB_CHKSUM

Enabling the PLL of the device not only allows to speed up the communication between the FlashRunner and the Power PC MCU, but also makes the target device much more performant when executing operations. This is even more relevant since we use the "*Standard Software Driver*" (aka microkernel), which is a set of pre-compiled functions supplied by silicon producers to operate on the device. Let us analyse how this can influence the standard commands:

- **Masserase**
  This command erases all the content of the memory selected. The supported memories for this command are: Code Flash, Data Flash, HSM Code, and HSM Data.
  Optionally, it is also possible to send four additional parameters to this command to select only some sectors of memory to be erased. This can be useful to reduce the erase time and it could be needed if the user wants to preserve some data into the memory and partially reprogram it. See the chapter "Sector Erase" for more information.
  The duration of Masserase depends only on the target device characteristics and there is not much that we can do to improve this. Typically, it is extremely long, even longer than all the other operations combined together.
  The suggestion is to skip this operation in case the device is virgin. This can be done using a conditional script based on the result of the blankcheck operation:

  ```
  #IFERR TPCMD BLANKCHECK F
  #THEN TPCMD MASSERASE F
  #THEN TPCMD BLANKCHECK F
  ```

- **BlankCheck**
  This command checks that all the bits of the selected memory are set to 1. The supported memories for this command are: Code Flash, Data Flash, HSM Code, and HSM Data. This operation is executed internally by the device and it is typically extremely fast.
  Optionally, it is also possible to send two additional parameters to this command: the address from where to start checking and the number of bytes to check.

  The duration of this operation depends on the size of the memory and by the internal frequency of the Power PC MCU. So, enabling PLL will reduce a lot the duration of this command and it can be about ten times faster.

- **Program**
  This command takes the customer's data from the FRB file and programs them into the selected memory, the supported memories for this command are: Code Flash, Data Flash, HSM Code, HSM Data, and UTEST.
  Optionally, it is also possible to send two additional parameters to this command: the address from where to start programming and the number of bytes to be programmed.

The duration of this command depends on many factors which are all important:

- The target device characteristics, in other words, how much time is needed to write data into the memory.
- The target device frequency because it determines how fast the device is to process data, so it is important to have the PLL enabled.
- The JTAG frequency because it sets the bitrate of the communication between FlashRunner and the target device.

The suggestion to improve the performance of this command is to set the FRB file with the "IGNORE_BLANK_PAGE" option. This will skip the program operation for any page which contains only 0xFF bytes.

```
#TPSETSRC myData.frb IGNORE_BLANK_PAGE
```

- **Verify Readout**
  This command checks that data contained in the memory of the device corresponds to FRB data. The supported memories for this command are: Code Flash, Data Flash, HSM Code, HSM Data, and UTEST.
  Optionally, it is also possible to send two additional parameters to this command: the address from where to start checking and the number of bytes to check.

```
#TPCMD VERIFY F R 0xFC0000 0x1000
```

This command works exactly like the program command with the only exception that the device reads (instead of writing) data from the memory and compares that with the data received from FlashRunner.
Since the mechanism is totally equal to the one used by the program command, if any error was introduced during the program command, it is possible that the same error could be introduced during the verify command and this could lead to a possible undetected error. For this reason, we suggest using the verify checksum instead of the verify readout, or maybe combining them.

- **Verify Checksum**
  This command asks the target device to calculate the 32-bit checksum of the selected memory region, meanwhile, the FlashRunner calculates the expected checksum according to FRB data and then the two values are compared. The supported memories for this command are: Code Flash, Data Flash, HSM Code, and HSM Data. This operation is executed internally by the device and it is typically extremely fast, almost like the blankcheck.

```
#TPCMD VERIFY F S
```

Optionally, it is also possible to send up to three additional parameters to this command: the address from where to start the checksum calculation, the number of bytes to consider in the calculation, and the expected checksum value. This will result faster because the FlashRunner does not need to spend time calculating the checksum of the FRB file.

```
#TPCMD VERIFY F S 0x00FC0000 0x00001000 0x4610E323
```

To get the value to use as the expected checksum parameter, you can use the command below, which can be executed by FlashRunner without being connected to the target device because it is just an internal calculation and it will return the commands to use in the real-time log.

```
#TPCMD CALC_FRB_CHKSUM
```

We also have an additional feature about checksum: some customers requested to return the checksum of the data when performing the verify checksum command and we gave them the possibility to enable this feature by setting the following parameter.

```
#TCSETPAR PRINT_CHKSUM_ON_VERIFY YES
```

This parameter does not give any benefits in terms of performance, in fact, this feature aims to give additional feedback to the customer's application which can verify that FlashRunner is working with the expected data. For example, running a project which has the checksum print enabled, the FlashRunner will return the following string when executing the verify checksum:

```
Start = 0x00FC0000, Size = 0x00100000, CheckSum = 0x4689B04
```

Note: the algorithm used to calculate the 32-bit checksum is the one defined by ST and NXP.

- **Margin check**
  This optional command can be used by customers to check that the voltage levels of all the memory cells are at the proper level. This is useful to guarantee the expected data retention. Basically, this command asks the device to calculate the MISR values for each bank of memory and for both threshold levels: high and low. Then the driver will check that MISR values obtained using a high threshold correspond to the ones obtained using a low threshold. Attention: this command is not verifying that the memory content is matching with the FRB data, it is just checking the memory status, so users should call this command after the verify command. This margin check command can simply be called without any parameters:

```
#TPCMD MARGIN_CHECK
```

Warning from ST and NXP: doing margin reads repetitively results in degradation of the flash memory array, and shortens the expected lifetime experienced at normal read levels. For these reasons the margin read usage is allowed only in factory mode, while it is forbidden to use it inside the user application.

After this explanation should be clear what can be improved by the user, what can be improved by SMH, and what cannot be improved because it depends only on the characteristics of the target device.

Warning: performing the program and verify commands using an FRB which does not contain any data for the selected memory region will return pass. The driver has been designed in this way to be more flexible so, basically, if a customer gives no data to program and verify, then the driver does not perform any operation and it just returns pass after completing the research for the data.

4. Sector erase

As already said in the previous chapter, the user can also choose to erase only some sectors of memory just by setting some more parameters to the masserase command. This could be the most optimized choice in case the erase cannot be skipped.

```
#TPCMD MASSERASE <MemType> <SEL0> <SEL1> <SEL2> <SEL3>
```

The parameters of this command are:

- **`MemType`**
  This indicates the memory chosen, it can be one of the following characters:
  - **`F`** – Code Flash memory.
  - **`E`** – Data Flash memory.
  - **`A`** – HSM Code memory.
  - **`B`** – HSM Data memory.

- **`SEL0 SEL1 SEL2 SEL3`**
  These are the bitmasks that select the block of memory to erase. It is forbidden to select a block that is not part of the **`MemType`** selected.

Usage example:

```
#TPCMD MASSERASE F 0x03DE0000 0x0 0x0 0x0
```

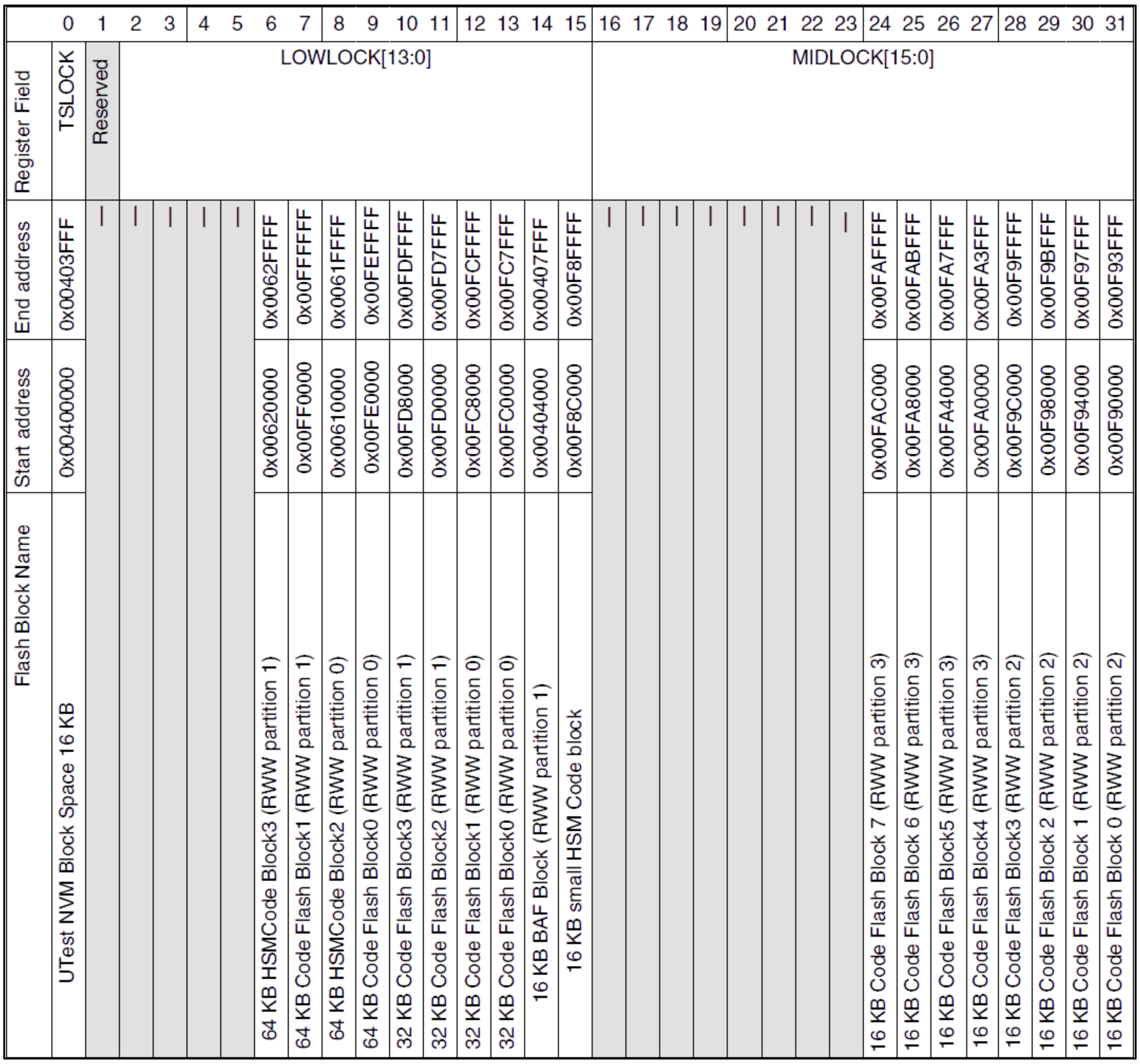
Using bitmasks rather than addresses is a design choice that we made to get higher reliability. In fact, the sectors have different sizes and they are not in a fixed order for all devices so, decoding addresses to bitmasks from the software is very complex and complexity does not go along with reliability. Moreover, also the API of ST and NXP work in this way, so we are following their choice as well.

To check that the operation was successfully done, it is possible to perform the blankcheck operation specifying the address and size, so the user will be able to check that the memory sectors were properly selected. For example:

```
#TPCMD BLANKCHECK F 0xFC0000 0x140000
```

The command to erase only some sectors of memory may not seem user-friendly at all but, with some explanations, it could be easier to understand. In fact, choosing the bits to select the sectors of memory to be erased requires a bit of effort and just by following the Reference Manual of the device. The trick is using the memory map, where the sectors are defined, in addition to the tables where the bits of the selectors are assigned to the sectors, this can be typically seen in the description of the "*PASS_LOCKn_PGn*" registers.

Here we are showing a practical example for an MPC5746C supposing that we want to erase only the first part of the code flash memory, from 0x00FC0000 to 0x00FFFFFF (0x40000 bytes). Reported below there is the bit map of the SEL0 register taken directly from the reference manual of the device.



	Register Field	End address	Start address	Flash Block Name
0	TSLOCK	0x00403FFF	0x00400000	UTest NVM Block Space 16 KB
1	Reserved	—		
2	LOWLOCK[13:0]	—		
3		—		
4		—		
5		—		
6		0x0062FFFF	0x00620000	64 KB HSMCode Block3 (RWW partition 1)
7		0x00FFFFFF	0x00FF0000	64 KB Code Flash Block1 (RWW partition 1)
8		0x0061FFFF	0x00610000	64 KB HSMCode Block2 (RWW partition 0)
9		0x00FEFFFF	0x00FE0000	64 KB Code Flash Block0 (RWW partition 0)
10		0x00FDFFFF	0x00FD8000	32 KB Code Flash Block3 (RWW partition 1)
11		0x00FD7FFF	0x00FD0000	32 KB Code Flash Block2 (RWW partition 1)
12		0x00FCFFFF	0x00FC8000	32 KB Code Flash Block1 (RWW partition 0)
13		0x00FC7FFF	0x00FC0000	32 KB Code Flash Block0 (RWW partition 0)
14		0x00407FFF	0x00404000	16 KB BAF Block (RWW partition 1)
15		0x00F8FFFF	0x00F8C000	16 KB small HSM Code block
16	MIDLOCK[15:0]	—		
17		—		
18		—		
19		—		
20		—		
21		—		
22		—		
23		—		
24		0x00FAFFFF	0x00FAC000	16 KB Code Flash Block 7 (RWW partition 3)
25		0x00FABFFF	0x00FA8000	16 KB Code Flash Block 6 (RWW partition 3)
26		0x00FA7FFF	0x00FA4000	16 KB Code Flash Block5 (RWW partition 3)
27		0x00FA3FFF	0x00FA0000	16 KB Code Flash Block4 (RWW partition 3)
28		0x00F9FFFF	0x00F9C000	16 KB Code Flash Block3 (RWW partition 2)
29		0x00F9BFFF	0x00F98000	16 KB Code Flash Block 2 (RWW partition 2)
30		0x00F97FFF	0x00F94000	16 KB Code Flash Block 1 (RWW partition 2)
31		0x00F93FFF	0x00F90000	16 KB Code Flash Block 0 (RWW partition 2)

We just need to set to 1 every bit which corresponds to a sector that we want to erase, for our example we need to set these bits: 7, 9, 10, 11, 12, and 13. Attention: the bit ordering used in the reference manual must be inverted, so the bit 7 in the table above corresponds to 2^{23}, and the result of setting those bits to 1 is: 0x017C0000. In conclusion, to erase and blanckcheck the code flash memory from 0x00FC0000 to 0x00FFFFFF of the MPC5746C, the following commands must be used:

```
#TPCMD MASSERASE F 0x017C0000 0x0 0x0 0x0
#TPCMD BLANKCHECK F 0xFC0000 0x40000
```

There are a couple of additional tricks that can be used to simplify the usage of this command. The first one is to look in the first part of the project where we define the default bitmask to completely erase any memory of the device. Below you can see the ones for MPC5746C, where we have three different values for SEL0 register (to erase code flash, or data flash, or HSM code), one value for SEL1 to erase HSM data, and one value for SEL2 to erase another part of the code flash.

```
#TCSETDEV SEL0_ERASE_CFLASH 0x017C0000
#TCSETDEV SEL0_ERASE_DFLASH 0x000000FF
#TCSETDEV SEL0_ERASE_HSMCODE 0x02810000
#TCSETDEV SEL1_ERASE_HSMDATA 0x00000003
#TCSETDEV SEL2_ERASE_CFLASH 0x000003FF
```

Some more tricks are:

- Ignoring SEL3 because it is typically set to zero, it could be different from 0 just for those SPC58xx which have 10MB for code flash memory.
- Considering the SEL2 only for code flash because other memory types are mapped only on SEL0 and SEL1.
- Code flash is often mapped also on SEL0 mixed with other memory types.

5. UTEST memory

UTEST is a very special one-time programmable (OTP) memory, which should never be programmed randomly until you want to trash the device. Since this memory is so critical, in this chapter we want to explain how to properly manage it using FlashRunner. These are the most important memory regions inside UTEST:

JTAG Password and PASS Password Groups

For the most recent devices, this region starts at 0x400120 and it contains (in order):

- JTAG Password [32 bytes]
- PASS password - Group 0 [32 bytes]
- PASS password - Group 1 [32 bytes]
- PASS password - Group 2 [32 bytes]
- PASS password - Group 3 [32 bytes]

Pay attention that these passwords cannot be read back after locking the device, the programmer will read all bytes as 0xFF, so the user must pay attention not to try to re-program these data because it will generate an error.

Life Cycle

This region determines the status of the device and, according to that, some features are enabled (like JTAG Password lock). There are two standards for this memory region depending on the device:

- **MPC57xx/SPC57xx**
  For these devices, mainly supplied by NXP (Freescale), the life cycles are 8 bytes long and they start at 0x400200. A single life cycle slot can be in one of the following states:
  - **INACTIVE**: `0x55AA50AF 0x55AA50AF`
  - **ACTIVE**: `0x55AA50AF 0xFFFFFFFF`
  - **ERASED**: `0xFFFFFFFF 0xFFFFFFFF`

- **SPC58xx**
  Instead, for these devices which are mainly supplied by ST Microelectronics, the life cycles are 16 bytes long and they start at 0x4001E0. The accepted states for a single life cycle slot are:
  - **INACTIVE**: `0x55AA50AF 0xFFFFFFFF 0x55AA50AF 0xFFFFFFFF`
  - **ACTIVE**: `0x55AA50AF 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF`
  - **ERASED**: `0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF`

To get the device in a valid working condition, there must be only one **ACTIVE** life cycle slot: all the previous ones must be **INACTIVE** and all the following ones must be **ERASED**.

DCF records

DCF records are the most tricky part of the UTEST because they work as a stack and they are not programmed at the same address defined by the user in the FRB, but they are programmed in the first available slot. The driver manages the stack checking the last valid DCF records and programming the new one at the bottom of the stack.
Since the user cannot know where the first slot available is, when creating the FRB, the DCF records must be inserted into the first address of this region. Usually, the start address is at 0x400308, 0x400320, 0x400348, or 0x400370, depending on the device because of the size of the reserved area after the DCF Start Record (0x400300) which is not standard and fixed.

DCF records are composed of a control word and a data word, in total it is and the data word is defined in the first 32 bits. In simple terms, the control word defines the pointer to a specific configuration and the data word defines the value to use.

An interesting peculiarity of DCF records is that they define a configuration and, if the user wants to change a configuration, he can do that just by adding a new DCF record to the stack. In fact, if two DCF records are defining the same setting, only the last one added is considered by the device. For example, this is feature is used to enable or disable censorship when reprogramming a locked device. Our algorithm is pretty smart because during the verification process it also checks that no conflicts are present and it will return an error in case it finds two DCF records with the same control word but different data words defined in the customer's firmware.
This is valid for the majority of DCF records, but there are some exceptions that FlashRunner is managing in a more sophisticated way according to the specifications, such as those DCF records that require triple-voting or that are order-dependent. Please, check the reference manual to understand more.

Attention: SPC58xx have a page size of 128-bit, so two DCF records must be programmed in a single operation. If there are an odd number of DCF records, the programmer will automatically duplicate the last one to fill all the remaining bytes of the page with valid data. This is very important because leaving those unprogrammed could cause problems and will not allow the programming of any additional DCF records.

Customer OTP data

This region is just an OTP region where the customer can write any data he needs. This is typically used for serial numbering.

Additional notes about UTEST

What we described before is valid for those devices which have 32-byte passwords; while the older devices which have 8-byte passwords are a bit different because they set the passwords through the DCF records.

As said at the beginning, UTEST memory is OTP, it cannot be erased or blank-checked, if the user tries to write some data into a non-blank space, he will get an error. Since into UTEST there are some data programmed by the silicon producer and some reserved regions, it is not possible to use the verify checksum method.

If the user only writes passwords into UTEST and then you perform a power-on-reset of the device, he will read all DCF records at 0x00. To solve this problem, the user must set the FlashRunner to unlock the device using the JTAG password, after that, all DCF records will be normally readable.

According to DCF records values, after locking a device, the user may not be able to read/write into HSM memory regions anymore.

As you can understand from the last two paragraphs, there are some very strange behaviours to consider and we also have limited knowledge about that. In case you notice any other “corner case” problem, we will analyse that with NXP or ST (if needed).

6. Lock and unlock using JTAG password

A very common usage case is programming and locking the device once and, in a second moment, wanting to reprogram something on the memory.

Lock

This topic is strictly related to what we explained about UTEST memory in the previous chapter. Basically to lock a Power PC MCU with the JTAG password you need to program the JTAG password and all the four pass groups, then you need to set the required DCF records and, in the end, you need to activate the proper life cycle slot. This procedure is only valid for the most recent devices that have 32-byte passwords, while the older devices which have 8-byte passwords are a bit different because they set the passwords through the DCF records.

To program these particular data you can simply put them inside the FRB or set them using dynamic memory, so you just need to call the program and verify commands.
Otherwise, there are also a couple of commands which may result easier (depending on your taste):

- **`SET_PASSWORDS <32bit_1> ... <32bit_8>`**
  Program (and verify) all the passwords with the same values, this means that the JTAG password and all the four "*PASS password – Group*" will be set with the values passed as parameters to this command. This command cannot be used after locking a device because the passwords cannot be read or modified anymore. This command is not supported for the older devices which have 8-byte passwords.

- **`SET_LIFECYCLE <start_addr> <level>`**
  Program (and verify) the life cycles to reach the indicated level. This may be a bit tricky, so let us explain a bit better:
    - **`<start_addr>`** must indicate the start address of the first life cycle slot.
    - **`<level>`** must indicate which is the life cycle slot to set as active (level = 0 is not allowed).

  For example, for SPC58xx, setting **`start_addr`** to 0x4001E0 and **`level`** to 4 will activate the "*In-Field*" state.

- **`SET_STD_DCF`**
  Program (and verify) a set of "standard" DCF records that will lock the device. This set of records includes all the DCF records for the LOCKx PASS GROUPx and the one for the censorship and they cannot be changed by the user. *Warning: this command is meant only for test purposes, it should never be used in production. It has been developed to help those customers which may need to do tests before receiving the ultimate firmware from their R&D department.*

Unlock, re-program and re-lock

To access a locked device the programmer must send the JTAG password during the execution of the connect command. The user must set the FlashRunner to use the JTAG password before calling that command and this can be done by setting this parameter to YES:

```
#TCSETPAR PASS_EN YES
```

After enabling the JTAG password, the user must send the value of the password to use to unlock the device and this can be done using dynamic memory (or FRB). Depending on the device, passwords can be 32-bytes long or 8-bytes long:

- For 32-byte passwords, they must be set at 0x00400120, which corresponds to the actual memory address of UTEST memory.
- For 8-byte passwords, they do not have a fixed address since they are defined through DCF records, so we use 0x00000000 as a fake address to define them.

Alternatively, it is also possible to use the command: **`PWD_ENTRY <32bit_1> ... <32bit_n>`**. Which sets the password to unlock the device and which must be called before the connect command. It can be used to set both 8-byte (two 32-bit words) and 32-byte (eight 32-bit words) passwords.

When unlocking a device using the JTAG password, due to a peculiar behaviour, the device cannot be properly reset. This implies that the programmed firmware may be running, so it is not possible to properly enable the PLL. For the same reason, after unlocking the device we strongly suggest not performing any operation on it but only calling the command **`CENSOR NO`**. This command will program and verify a DCF record to disable the JTAG password so, after that, the FlashRunner can simply access to the device without any password: the device can be properly reset, the PLL can be enabled and the memory can be reprogrammed without any risk of interference by a running firmware. When all operations are completed, it is enough to call the command **`CENSOR YES`** to restore the censorship with the JTAG password.

A couple of more words about this fundamental command:

`CENSOR <YES | NO>`
Enable or disable JTAG password by programming (and verifying) the proper DCF record. This cannot be done without limits because every time the status changes a new DCF record is added to the stack and this memory region of the device is limited. Anyway, the limit is much higher than what is needed, in fact, it is possible to change status about 1000 times, while typically it is only needed a few times.

Case study

In this section, we will show how the operations described before can be applied to an SPC584C80. Starting from a virgin device, using the project below, we are programming all the memories of the device (code flash, data flash, HSM code, and HSM data), then we are programming passwords into the UTEST, then we are programming DCF records and, as the last step, we are setting the status of the life cycle to “*In field*”.
In the project, we are using *myData.frb* which is containing all the data to be programmed and also a “*variable data zone*” of 160 bytes at 0x400120 to accept data from dynamic memory.

```
!ENGINEMASK 0x00000001
#LOADDRIVER libPowerMCU.so STMICROELECTRONICS SPC58 SPC584C80
...
#TCSETDEV SEL0_ERASE_CFLASH 0x03DE0000
#TCSETDEV SEL0_ERASE_HSMCODE 0x0C200000
#TCSETDEV SEL0_ERASE_HSMDATA 0x00000003
#TCSETDEV SEL1_ERASE_DFLASH 0x0000000F
#TCSETDEV SEL2_ERASE_CFLASH 0x0000FFFF
#TCSETDEV SWT_BASE 0xF4058000
!CRC 0x0E867CB6
#TCSETPAR PASS_EN NO
#TCSETPAR PLL_EN YES
#TCSETPAR PROTCLK 37500000
#TCSETPAR PWDOWN 100
#TCSETPAR PWUP 100
#TCSETPAR RSTDOWN 100
#TCSETPAR RSTDRV OPENDRAIN
#TCSETPAR RSTUP 100
#TCSETPAR VPROG0 5000
#TCSETPAR CMODE JTAG
; It is better to always clear dynamic memory before using that
#DYNMEMCLEAR
; Set passwords through dynamic memory
#DYNMEMSET2 0x400120 16 000102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x400130 16 101112131415161718191A1B1C1D1E1F
#DYNMEMSET2 0x400140 16 100102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x400150 16 101112131415161718191A1B1C1D1E1F
#DYNMEMSET2 0x400160 16 200102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x400170 16 101112131415161718191A1B1C1D1E1F
#DYNMEMSET2 0x400180 16 300102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x400190 16 101112131415161718191A1B1C1D1E1F
#DYNMEMSET2 0x4001A0 16 400102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x4001B0 16 101112131415161718191A1B1C1D1E1F
; Set the FRB file which is containing all data
#TPSETSRC myData.frb IGNORE_BLANK_PAGE
```

```
#TPSTART
#TPCMD CONNECT
; Program Code Flash memory
#IFERR TPCMD BLANKCHECK F
#THEN TPCMD MASSERASE F
#THEN TPCMD BLANKCHECK F
#TPCMD VERIFY F R
#TPCMD VERIFY F S
; Program Data Flash memory
#IFERR TPCMD BLANKCHECK E
#THEN TPCMD MASSERASE E
#THEN TPCMD BLANKCHECK E
#TPCMD VERIFY E R
#TPCMD VERIFY E S
; Program HSM Code  memory
#IFERR TPCMD BLANKCHECK A
#THEN TPCMD MASSERASE A
#THEN TPCMD BLANKCHECK A
#TPCMD VERIFY A R
#TPCMD VERIFY A S
; Program HSM Data memory
#IFERR TPCMD BLANKCHECK B
#THEN TPCMD MASSERASE B
#THEN TPCMD BLANKCHECK B
#TPCMD VERIFY B R
#TPCMD VERIFY B S
; Program UTEST memory (Password and DCF records)
#TPCMD PROGRAM U
#TPCMD VERIFY U R
; Set life cycle to "In Field"
#TPCMD SET_LIFECYCLE 0x4001E0 4
#TPCMD DISCONNECT
#TPEND
```

After a power-on-reset of the device, the censorship will be active and the JTAG password will be required to access to the memory of the device.
Please, note these kinds of projects are too complex to be generated automatically by the Workbench, so the user must manually insert most of these commands using the project editor.

Let us suppose that now we want to re-program the first part of the code flash memory, as explained before, doing that is not as easy as saying that because of the complexity of the device. In the following project, we are showing the complete procedure to achieve that.

```
!ENGINEMASK 0x00000001
#LOADDRIVER libPowerMCU.so STMICROELECTRONICS SPC58 SPC584C80
...
#TCSETDEV SEL0_ERASE_CFLASH 0x03DE0000
#TCSETDEV SEL0_ERASE_HSMCODE 0x0C200000
#TCSETDEV SEL0_ERASE_HSMDATA 0x00000003
#TCSETDEV SEL1_ERASE_DFLASH 0x0000000F
#TCSETDEV SEL2_ERASE_CFLASH 0x0000FFFF
#TCSETDEV SWT_BASE 0xF4058000
!CRC 0x0E867CB6
; Enable JTAG password and disable PLL
#TCSETPAR PASS_EN YES
#TCSETPAR PLL_EN NO
...
; Set the FRB file which is containing the new data
#TPSETSRC myNewData.frb IGNORE_BLANK_PAGE
#DYNMEMCLEAR
; Set the JTAG password through dynamic memory
#DYNMEMSET2 0x400120 16 000102030405060708090A0B0C0D0E0F
#DYNMEMSET2 0x400130 16 101112131415161718191A1B1C1D1E1F
#TPSTART
#TPCMD CONNECT
; Disable censorship
#TPCMD CENSOR NO
; Power-on-reset the device (disable password and enable PLL)
#TCSETPAR PASS_EN NO
#TCSETPAR PLL_EN YES
#TPCMD RECONNECT
; Re-program the first part of the code flash
#IFERR TPCMD BLANKCHECK F 0x00FC0000 0x140000
#THEN TPCMD MASSERASE F 0x03DE0000 0x0 0x0000001F 0x0
#THEN TPCMD BLANKCHECK F 0x00FC0000 0x140000
#TPCMD PROGRAM F R 0x00FC0000 0x140000
#TPCMD VERIFY F R 0x00FC0000 0x140000
#TPCMD VERIFY F S 0x00FC0000 0x140000 0x12345678
; Re-enable censorship
#TPCMD CENSOR YES
#TPCMD DISCONNECT
#TPEND
```

7. Register operations

Sometimes customers need to do special operations on their devices and they may need to directly operate on some registers. That is why we added these three commands which allow accessing to any register of the device:

- **`WRITE_REG <registerAddr> <value>`**
  Write register command: write the value into the selected register.

- **`VERIFY_REG <registerAddr> <value>`**
  Verify register command: verify that the value of the selected register corresponds to the one requested.

- **`READ_REG <registerAddr>`**
  Read register command: read the value of the selected register.

These commands are typically not needed and they should not be used unless you really know what you are doing because they are used just for very special operations. These particular commands allow the customers to read, write and verify the value of a volatile register of the MCU. For example, they can be used to disable an internal watchdog that is not automatically disabled by FlashRunner.

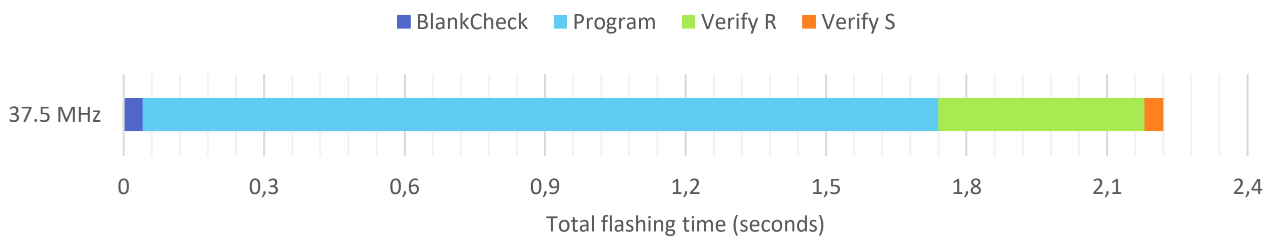
Do not hesitate to ask for support in case of doubts because operating on registers could be critical.

8. Flashing time examples

These are some examples of flashing times using our new drivers and OS > 3.00.

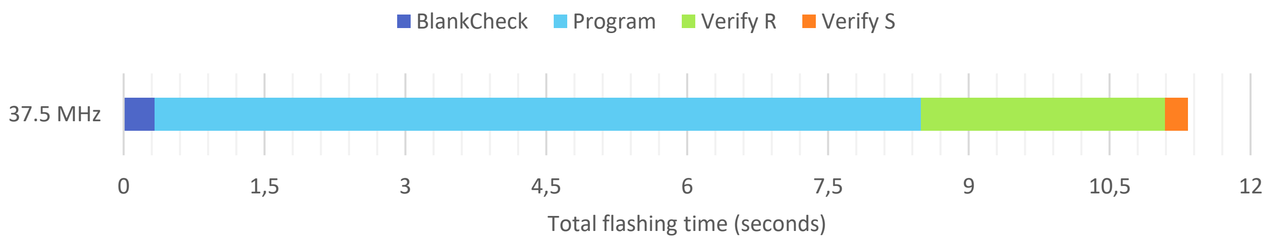
Additional notes:
1. Times were measured using FRB files containing random data to cover the entire code flash memory of the target device.
2. We do not consider the masserase time in the total flashing time because it is usually skipped.
3. Verify checksum executed sending the pre-calculated checksum as a parameter.

SPC582B60 [1 MB]



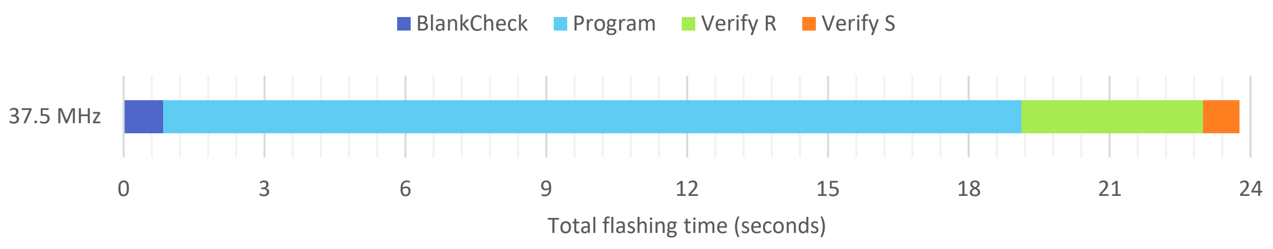
JTAG Freq.	Masserase	BlankCheck	Program	Verify R	Verify S
37.5 MHz	6.94 s	0.04 s	1.70 s	0.44 s	0.04 s

SPC58NE84 [6 MB]



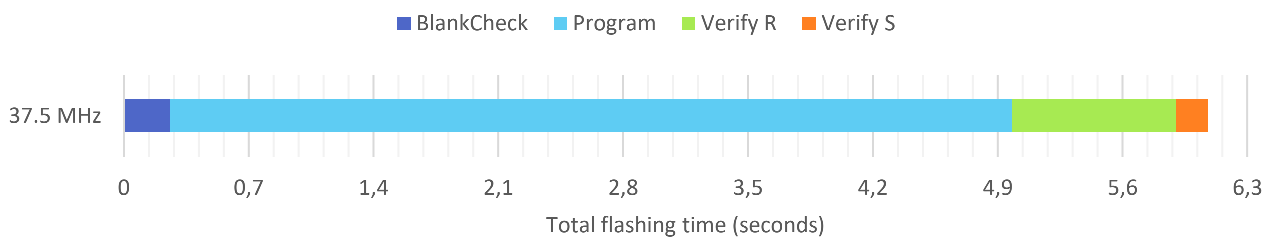
JTAG Freq.	Masserase	BlankCheck	Program	Verify R	Verify S
37.5 MHz	27.45 s	0.33 s	8.16 s	2.60 s	0.24 s

MPC5777C [8 MB]



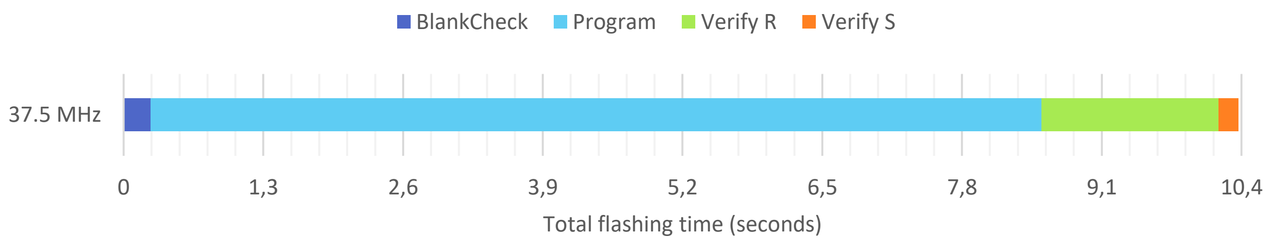
JTAG Freq.	Masserase	BlankCheck	Program	Verify R	Verify S
37.5 MHz	30.71 s	0.84 s	18.28 s	3.86 s	0.78 s

S32R274 [1.91 MB]



JTAG Freq.	Masserase	BlankCheck	Program	Verify R	Verify S
37.5 MHz	8.44 s	0.26 s	4.72 s	0.92 s	0.18 s

MPC5747G [3.75 MB]



JTAG Freq.	Masserase	BlankCheck	Program	Verify R	Verify S
37.5 MHz	14.10 s	0.25 s	8.29 s	1.65 s	0.18 s

Other flashing time examples

These are some flashing times of other devices. Time for masserase is not reported because it is typically skipped for virgin devices and total time is considering both verify readout and checksum.

Device	Size [JTAG Freq.]	BlankCheck	Program	Verify R	Verify S	Total
MPC5746C	3 MB [37.5 MHz]	0.14 s	6.23 s	1.20 s	0.09 s	7.66 s
MPC5748G	5.75 MB [37.5 MHz]	0.39 s	12.95 s	2.50 s	0.27 s	16.11 s
MPC5744P	2.4 MB [37.5 MHz]	0.12 s	5.19 s	1.05 s	0.09 s	6.45 s
MPC5746R	3.9 MB [37.5 MHz]	0.25 s	8.52 s	1.70 s	0.18 s	10.65 s
MPC5775B	4 MB [37.5 MHz]	0.34 s	2.22 s	0.62 s	0.08 s	3.26 s
SPC570S50	0.5 MB [37.5 MHz]	0.02 s	0.88 s	0.22 s	0.02 s	1.14 s
SPC572L64	1.5 MB [37.5 MHz]	0.09 s	2.47 s	0.67 s	0.07 s	3.30 s
SPC584B70	2 MB [37.5 MHz]	0.12 s	2.71 s	0.88 s	0.07 s	3.78 s
SPC584C80	4 MB [37.5 MHz]	0.17 s	1.34 s	0.44 s	0.02 s	1.97 s
SPC58NN84	6 MB [37.5 MHz]	0.33 s	7.81 s	2.63 s	0.24 s	11.01 s
SPC58NH92	10 MB [37.5 MHz]	0.62 s	13.62 s	4.37 s	0.47 s	19.08 s

9. Frequently Asked Questions

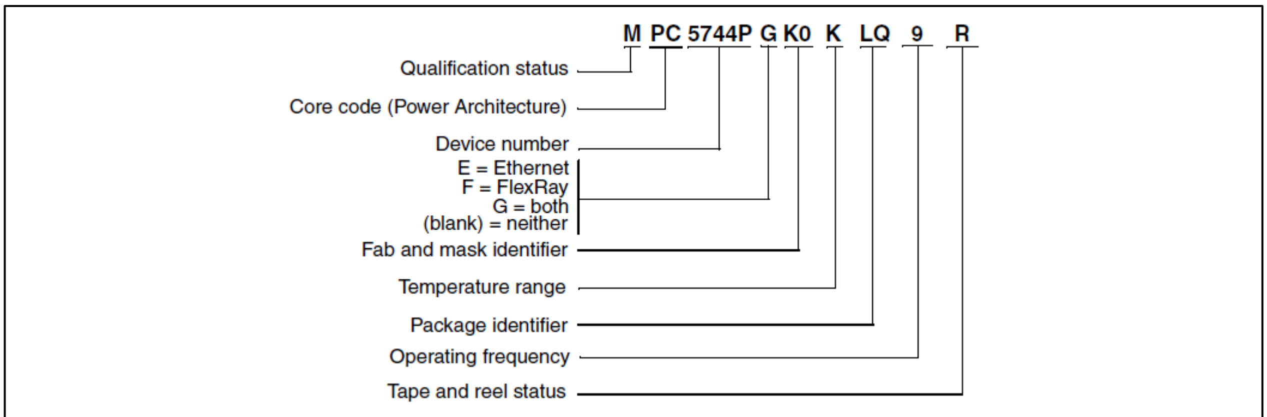
Some answers to the most frequently asked questions:

Are these new updates available for all FlashRunner models?

No, these updates are only for the *FlashRunner NXG*, the *FlashRunner 2.0*, and the *FlashRunner HS*. These newer models are more powerful and they can be much faster than the FlashRunner of the classic series.

What are the differences between MPC57xx, PPC57xx, and SPC57xx?

From our point of view, there are no significant differences. NXP (Freescale) uses this particular standard for their Power PC MCUs, where the first letter indicates the qualification, as you can see in the example below. Since we just do in-system programming, for us they are equals and that letter is irrelevant. That is why in our device list all three versions with different qualifications are merged and named as MPC57xx.



Temperature range	Package identifier	Operating frequency	Qualification status	Tape and reel status
M = –40°C to +125°C	LQ = 144 LQFP	9 = 200 MHz	P = Pre-qualification	R = Tape and reel
K = –40°C to +135°C for extended temp (+165°C T_J)	MM = 257 MAPBGA	8 = 180 MHz	M = Fully spec. qualified, general market flow	(blank) = Trays
		5 = 150 MHz	S = Fully spec. qualified, automotive flow	

Note: Not all options are available on all devices.

What about SPC56xx and MPC56xx?

We are also supporting them on a different driver because they have different characteristics. Find more details on their dedicated document: Interfacing FlashRunner with Power PC MCUs [56].

How do you support S32R294?

This new radar MCU from NXP is extremely powerful and it is a bit different compared to the other Power PC MCUs that we wrote about in this document. In fact, this MCU does not have an internal flash, but it uses an external serial NOR flash in Quad-SPI, this means that we developed an algorithm that allows customers to program the external memory through the MCU.
Basically, this is totally transparent to the user and the "tips and tricks" explained before are still valid. This device, also known as RaceRunnerUltra2, has a special OTP area called eFuse which is very critical and that, of course, we are already supporting.

DCF records – Migrating from the traditional FR line to FR 2.0 platform

A lot of our customers are migrating from the traditional line of FlashRunner to the 2.0 platform because the performance is much better and also because FlashRunner 2.0 is including many more features and more devices supported, especially for these PowerPC microcontrollers described in this document. Just consider that most of these microcontrollers are more powerful than the MCU used inside the FlashRunners of the traditional line, while this is not valid at all for FlashRunner 2.0 which is basically a small computer with a lot of RAM and Linux inside.

Anyway, a very common question coming when doing this migration is how to manage DCF records because their management has been implemented in different ways. This is an example of how a single DCF record is programmed and verified on the traditional line of FlashRunner:

```
TPCMD PROGRAM DCF $000055AA $001000B0
TPCMD VERIFY DCF $000055AA $001000B0
```

The first value corresponds to the data word and the second to the control word. So to move this DCF record on FlashRunner 2.0 you need to create a binary file containing these data in the same order and then create an FRB including that binary file.

Troubleshooting

These are the explanations for some common errors that the customer can try to fix by himself:

- **`Wrong id Code. Read 0x********, instead of 0x********`**
  This is the error that happens when the communication between the FlashRunner and the target MCU is not working at all. This error can be caused by wrong connections, selecting the wrong voltage level, not powering up the MCU or (not very often) using a protocol frequency that is too high.

- **`Error entering in debug mode: check connections, VPROG0 value and JTAG password`**
  This error has a clear description of what could be the aspects to check, especially the JTAG password is the typical cause for this error in case you locked the device.

- **`Debug status error`**
  This is a bad error because it could be an indication of a bricked device, maybe bad data was programmed on the UTEST memory. Anyway, don't lose faith and, first of all, double-check JTAG password. If it is ok, then check connections, power supply and any possible external reset sources.

- **`Error initialising RAM address 0x********`**
  This error commonly happens when using a protocol frequency that is too high on a hardware setup that is not optimal. In fact, when this happens at a frequency above 20 MHz, the FlashRunner is suggesting "**`Please, try using a lower frequency for JTAG protocol`**".
  This error can also occur due to an external reset, typically coming from a PMIC or an SBC.

- **`Error writing the microkernel into the RAM`**
  This error is quite similar to the previous one, so you should check the same factors.

- **`SSD microkernel returned an unidentified code: 0x********`**
  This error is the typical error that happens when an external reset is interfering with the flashing operations. Check if there is a reset pulse coming from a PMIC or an SBC and if the power supply is stable.

- **`Error executing instruction`**
- **`Timeout executing instruction`**
  These errors are quite similar to the previous one, so you should check the same factors.

Anyway, if you still need support, feel free to send us an email: Support@SMH-Tech.com