DC11563
Driver 2.32
September 2023
R. Ertolupi

Interfacing FlashRunner with Serial NOR Flash Memories

1. Introduction

This document aims to explain the changes, the improvements, and the new features of our new drivers for SPI NOR memories: ***SerMem4***, ***SerMem8*** and ***SerMem2x4***.
As you may know, these devices are extremely common and they are often the ones which determine the flashing time of a panel. So, being able to optimize operations for them, means that we can reduce the flashing time for most of our customers' applications.

The reason why we started this activity was initially just to support those new memories with the Octo-SPI interface and, since Quad-SPI is a sub-set of Octo-SPI, we decided to extend the new features and improvements also to all the SPI memories.
During this development, we worked together with silicon producers to take care of the small details about these devices. Even if it's true that most of them are meant to be compatible and exchangeable, but going deep into the configurations and features they are not compatible at all and we had to manage those peculiarities to exploit the full potential of any device.

Basically, these new drivers are supporting any SPI, Quad-SPI, Twin-Quad-SPI and Octo-SPI memory from the most popular silicon producers, such as: **Renesas** (Adesto / Dialog Semiconductor), **GigaDevice**, **Infineon** (Cypress / Spansion), **ISSI**, **Macronix**, **Micron**, **Winbond** and many others. We also joined the Xccela consortium, an association founded by Micron which promotes the Xccela bus as an open standard.

You can download the latest version of this document from this static link: Interfacing FlashRunner with Serial NOR Flash Memories.

2. Contents

3. How do serial NOR flash memories work?

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. These are the standard commands:

- **Masserase**
  This command erases all the content of the serial NOR flash memory (all bits are set to 1). Optionally, it is also possible to send two additional parameters to this command: the address from where to start erasing and the number of bytes to erase. Then the driver will automatically manage the sectors to do that operation as fast as possible according to user's requests. Usually, 4KB is the minimum sector size allowed for this operation.

  The duration of this operation depends only on the target device characteristics and there is not much that we can do to improve this. For some devices it could be extremely long, even longer than all the other operations together. For very few devices, this operation may be shorter according to the amount of data programmed in the memory.
  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 reads all the content of the serial NOR flash memory and checks that all bits are set to 1. This operation is terminated instantly when a 0 is found.
  Optionally, it is also possible to send two additional parameters to this command: the address from where to start reading and the number of bytes to check.

  The duration of this operation depends only on bitrate that has been set by the user. The bit rate directly depends on the protocol and by the frequency, so the suggestion is to use the highest frequency and the fastest protocol. See the examples below which show the time estimation for the blankcheck operation over a 512Mbit serial NOR flash memory:

Protocol	Frequency	Bitrate	Time for 512Mbit
SPI	1 MHz	1 Mbps	512 s
Quad-SPI SDR	25 MHz	100 Mbps	5.12 s
Octo-SPI SDR	50 MHz	400 Mbps	1.28 s
Octo-SPI DDR	30 MHz	480 Mbps	1.07 s

- **Program**
  This command takes the customer’s data from the FRB file and programs them into the memory.
  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 mainly depends on the target device but also on the bitrate. In fact, the FlashRunner sends the data of a single page (typically 256 bytes) to the device and then it must wait for the page to be programmed. So, having a higher bitrate only reduces the “send page” time, which usually means saving just 20% or 30% of the command time or even less depending on the target device timings. See the section “Other flashing time examples” to better understand this point.
  The suggestion to improve the performances 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**
  This command reads all the content of the serial NOR flash memory and checks that it corresponds to FRB data. This operation is terminated instantly when a mismatch is found.
  Optionally, it is also possible to send two additional parameters to this command: the address from where to start reading and the number of bytes to check.

  The duration of this operation should be the same as blankcheck, but with a very small delay because we need to manage and check data. When using the new drivers with an OS > 3.00, this delay is almost irrelevant.
  To improve the duration of this command, follow the same suggestions proposed for blankcheck and program commands.

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. Protocols and frequencies supported

With the new drivers we ampliated the list of supported protocols: we improved the support for Quad-SPI, added the support for Octo-SPI, and we are also giving the possibility to use DDR for some memories. Let us see in detail what this means and how it can be used.

First of all, we must know that the protocol defines that messages are composed of three elements: command, address, and data. These three elements can be transmitted differently, these are all the configurations supported by our new drivers:

- Command, address, and data using a single line (**1-1-1**). This is the standard SPI protocol.
- Command and address using a single line and data using four lines (**1-1-4**). This is called Extended-SPI, Quad-SPI, or just Quad.
- Command, address, and data using four lines (**4-4-4**). This is typically called QPI.
- Command and address using a single line and data using eight lines (**1-1-8**). This is called Extended-SPI, Octo-SPI, or Octal.
- Command, address, and data using eight lines (**8-8-8**). This is typically called OPI.

The names used for the configurations above are not always common between the various silicon producers, that is why we preferred to simplify the choice to the user between these three options:

1. **SPI**
   This is just the standard SPI (**1-1-1**).

2. **Quad-SPI**
   This includes both options which use four lines for data (**1-1-4**) and (**4-4-4**). Since there are no relevant performance differences between them, we hid this choice to the user and to let the driver decides which is the best option to use according to the device characteristics (not all devices offer both options).

3. **Octo-SPI**
   This includes both options which use eight lines for data (**1-1-8**) and (**8-8-8**). About this design choice, the considerations are the same as Quad-SPI.

Moreover, we added the support for **DDR** (Double Data Rate), also said DTR (Double Transfer Rate), which allows us to double the bitrate without doubling the frequency of the protocol clock. In fact, when using SDR (Single Data Rate), signals are sampled only on the rising edge of the clock, while, for DDR, signals are sampled on both rising and falling edges. This feature is not available for all devices.

In case the chosen device does not support DDR and the hardware setup does not support Quad-SPI or Octo-SPI, there is still the possibility to increase the bitrate by choosing a higher frequency and this has also been improved. When using the new drivers with an OS > 3.00, also these frequencies higher than 25.0 MHz can be selected: 27.27MHz, 30.00MHz, 33.33MHz, and 37.50MHz.
Attention: if you notice some instabilities when rising frequency or when trying to use Quad-SPI or Octo-SPI, it could be related to the hardware setup (see the next chapter).

5. Hardware setup

Just like an F1 car, which differently performs according to the weather and track conditions, the FlashRunner requires a good hardware setup to reach the best performances. In fact, the quality of the connections between the FlashRunner and the target board is extremely important and in this chapter we want to define some guidelines.

The **standard SPI** protocol specifies four fundamental signals: *Chip Select*, *Clock*, *MOSI* (Master Output Slave Input), and *MISO* (Master Input Slave Output). Serial NOR flash memories also have two more signals which need to be managed: *WP* (Write Protect) and *Hold*. These two lines must be kept high while operating on the target device using SPI protocol and, in case the customer's board does not allow them to be set high, then there may be a design problem.

The **Quad-SPI** used by serial NOR flash memories uses the same connections of standard SPI, it just changes the function of these four lines which are used as data lines:

- *MOSI* becomes *IO0*
- *MISO* becomes *IO1*
- *WP* becomes *IO2*
- *Hold* becomes *IO3*

So, there is no needs to change wirings to use Quad-SPI instead of SPI.

The **Octo-SPI** protocol is similar to Quad-SPI, just with four additional data lines: *IO4*, *IO5*, *IO6*, and *IO7*. This means that the backward compatibility with SPI is still maintained.

As explained in the previous chapter, using Quad-SPI or Octo-SPI gives a big advantage because the bitrate can be much higher, but there is also a disadvantage: a good hardware setup is required. In fact, while SPI is very robust and can work also in bad conditions, Quad-SPI and Octo-SPI are more sensitive because they have many synchronous data lines and the quality of the wirings is crucial:

- Use our flat cables with cable interface to go as near as possible to the target device.
- The wiring should be as short as possible and all the wiring should have all the same length.
- There should be as few discontinuities as possible (i.e. prefer one 30 cm cable instead of two 15 cm cables connected together).
- Add more ground wires between FlashRunner and the target device, possibly twisted with each signal line, especially with the clock signal.

Another common issue that can arise when multiple lines move at the same time is the crosstalk: the phenomenon by which signals transmitted on one or more lines generate undesired effects on other lines due to electromagnetic coupling between them. For example, when transmitting 0x00 followed by 0xFF in Octo-SPI, the simultaneous switching of eight data lines from 0 to 1 can pull the clock line to 1 and generate a spurious clock pulse.

To mitigate this effect, one possible solution is to add small resistors (i.e. 100 Ohm for 3.3V devices or 56 Ohm for 1.8V devices) in series on all data lines. These resistors decrease the current flow on data lines and then reduce the interaction effect on the clock line.

One more useful trick to protect the clock line from the crosstalk is to add a small capactitor (i.e. about 20~30 pF) between the clock and the ground line on the fixture side.

6. Register operations

Compared to the older version of SerMem, in the new drivers we extended the support for read/write register operations. We added these three commands: **`WRITE_REG`**, **`VERIFY_REG`**, and **`READ_REG`**. These commands are very flexible and they change their functionality according to the target device characteristics.

For ***SerMem4*** and ***SerMem2x4***, these are the command descriptions:

- **`WRITE_REG <registerName> <value>`**
  Write register command: write the value into the selected register.
- **`VERIFY_REG <registerName> <value>`**
  Verify register command: verify that the value of the selected register corresponds to the one requested.
- **`READ_REG <registerName>`**
  Read register command: read the value of the selected register.

The **`registerName`** parameter must be chosen from the tables below according to the family of the target device. For the non-listed families, only the status register (SR) is supported.

Device Family	Register Name	Description
AT25, **GD25** and **W25**	SR	Status Register
	SR2	Status register 2
	SR3	Status register 3
GD25_Gen2	SR	Status Register
	NVCR[0~7]	Non-volatile configuration register from 0 to 7
	VCR[0~7]	Volatile configuration register from 0 to 7
IS25_P and **IS25D_P**	SR	Status Register
	NVRR	Non-volatile read register
	VRR	Volatile read register
MT25Q, **MT25T** and **N25Q**	SR	Status Register
	NVCR	Non-volatile configuration register (2-byte)
	VCR	Volatile configuration register
S25FL-S, **S79FL-S**, **MX25** and **MX66**	SR	Status Register
	CR	Configuration register
S25HL-T, **S25HS-T**, **S25FL-L**, **S25FS-S** and **S70FS-S**	SR	Status Register
	CR1NV ~ CR4NV	Non-volatile configuration register from 1 to 4
	CR1V ~ CR4V	Volatile configuration register from 1 to 4
XT25	SR	Status Register
	SR1	Status register 1

For ***SerMem8***, the commands are a bit different because devices have different characteristics:

- **`WRITE_REG <registerName> <address> <value>`**
  Write register command: write the value into the selected register. The address parameter is needed only for some particular registers.
- **`VERIFY_REG <registerName> <address> <value>`**
  Verify register command: verify that the value of the selected register corresponds to the one requested. The address parameter is needed only for some particular registers.
- **`READ_REG <registerName> <address>`**
  Read register command: read the value of the selected register. The address parameter is needed only for some particular registers.

The **`registerName`** parameter must be chosen from the tables below according to the family of the target device:

Device Family	Register Name	Description
ATXP	SR	Status register
	SR2	Status register 2
	SR3	Status register 3
	IOCR	I/O Pin Drive Strength Control Register
GD25X, **IS25_X** and **MT35X**	SR	Status register
	NVCR [Addr]	Non-volatile configuration registers (address required)
	VCR [Addr]	Volatile configuration registers (address required)
MX25_W, **MX25_M**, **MX66_W**, and **MX66_M**	SR	Status register
	CR	Configuration register
	NVCR2 [Addr]	Non-volatile configuration registers 2 (address required)
	VCR2 [Addr]	Volatile configuration registers 2 (address required)
S28HS_T and **S28HL_T**	SR	Status register
	AR	Any register (address required)

Note: the address parameter must be taken from the documentation of the target device.

7. Verify using CRC

As you may know, CRC is a powerful tool widespread in many applications nowadays, mostly to quickly compare big amounts of data and detect changes.
CRC is the acronym of "Cyclic Redundancy Check" and it is an algorithm that calculates a code based on the data and their position. It is specifically designed to protect against accidental communication error, where it can provide quick and reasonable assurance of the integrity of data.
However, this algorithm has some limits because it uses a surjective function, associating to a single CRC value more combinations of data. This means that it is not suitable for protecting against intentional alteration of data, we underlined "intentional" because it is extremely difficult and highly improbable to get the same CRC from similar data.

We added this feature for those devices which have a built-in CRC calculator, such as the Micron's devices (MT25Q and MT35X) and some Infineon's devices (S25HS-T, S25HL-T, S28HS-T, and S28HL-T). Infineon implements a CRC-32 function which means it is using a 32-bit value to express the CRC value (4,294,967,296 different possible values). While Micron implements a CRC-64 function which means it is using a 64-bit value to express the CRC value (18,446,744,073,709,551,616 different possible values). It should be clear that having more bits corresponds to have higher assurance.

With our new drivers, the customer can choose to use these features to verify that the CRC of the memory content corresponds to the one of the expected data. This verify method is typically faster than reading out all the data from the memory, it could result slower just if the amount of data is not too big or if the selected bitrate is very high.
The user can choose between one of the three methods below to invoke the verify CRC command:

- **`VERIFY F S`**
  This command calculates the CRC of the FRB file and compares it with the one calculated by the device. Usage example:

  ```
  #TPCMD VERIFY F S
  ```

- **`VERIFY F S <expectedCRC>`**
  This command takes the pre-calculated CRC value and compares it with the one calculated by the device. This may result faster because the FlashRunner does not need to spend time calculating the CRC of the FRB file. Usage example:

  ```
  #TPCMD VERIFY F S 0xEE2D496C36202742
  ```

- **`VERIFY F S <startAddress> <size> <expectedCRC>`**
  This command is equivalent to the previous one, but it can be applied to a portion of the memory. Usage example:

  ```
  #TPCMD VERIFY F S 0x00000000 0x04000000 0x3E03A198
  ```

To get the value to use as expected CRC parameter, the customer can use one of the following commands:

- **`CALC_FRB_CRC32`**
- **`CALC_FRB_CRC64`**

These commands can be executed by FlashRunner without being connected to the target device because it is just an internal calculation and they will return the commands to use in the real-time log. Usage example:

```
...
#TPSETSRC myFirmware.frb
#TPSTART
#TPCMD CALC_FRB_CRC64
#TPEND
```

We also decided to use CRC to perform the blankcheck for these devices because it resulted faster in most of cases. This is completely hidden to the user because calling the standard command for the blankcheck operation, it will be automatically redirected to a CRC check.

If the customer still prefers to use the classic blankcheck method by reading out all data, it is possible to use blankcheck commands with the address and size parameters and selecting the entire memory space. Example to use the classic blankcheck instead of using CRC for a 512Mbit memory:

```
#TPCMD BLANKCHECK F 0x00000000 0x04000000
```

We also have an additional feature about CRC: some customers requested to return the CRC of the data when performing the verify readout command and we gave them the possibility to enable to this feature by setting one of the following parameters (the first one for CRC32 and the second one for CRC64).

```
#TCSETPAR PRINT_CRC32_ON_VERIFY YES
```

```
#TCSETPAR PRINT_CRC64_ON_VERIFY YES
```

These parameters do not give any benefits in terms of performances, on the contrary, they will cause a bit of delay during verify readout execution because they add the calculation of the CRC value. 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 CRC32 print enabled and verifying the first 0x00800000 bytes of the memory, the FlashRunner will return the following string:

```
    Start = 0x00000000, Size = 0x00800000, CRC32 = 0x57889B04
```

Note: algorithms used to calculate CRC32 or CRC64 are the ones respectively defined by Infineon and Micron.

8. Multiple memories on the same bus

Sometimes it happens to have more than a single memory on the same SPI bus different CS for anyone of them. This could happen because it is the device which is actually containing two memories, for example S70FL01GS is containing two S25FL512S, but this could be also a design choice of the customer.

In any case, we improved the flexibility of our new drivers to be able to manage up to three chips on the same bus with a single ISP channel. Before this optimization, it was necessary to connect a single ISP channel to each one of the serial NOR flash memories and to operate on them sequentially while, now, it is possible to manage them in parallel and this means saving a lot of time.

At the moment, in our list of the supported devices there are just two devices that use this special feature: the S70FL01GS, as mentioned before, and the MT35XU512_x3, which is a customization done for a customer who has three MT35XU512 on the same bus.
Using this new feature, we could improve the flashing time of about 70% for both of them.

This feature is available only for those devices which require it by default (like S70FL01GS) or if it is explicitly requested by the customer (like MT35XU512_x3).

9. Twin memories (SerMem2x4)

Most of the serial NOR flash memories on the market use SPI and Quad-SPI; recently the Octo-SPI memories are becoming popular as well, but the transition has not been so sharp. In fact, on the market there are also some memories called “Twin”: they are a hybrid solution between Quad-SPI and Octo-SPI memories. These memories are defined as “Twin” because they internally contain two independent Quad-SPI memories which only share the CS and CLK lines. The result is that they have the same pinout of an Octo-SPI memory and, so, they allow a higher bitrate compared to a standard Quad-SPI memory and they are also faster than Octo-SPI memories because they can execute two commands in parallel. However, being a hybrid solution, they also have some complications since two memories must be managed in parallel.

In our catalog, we have the ***SerMem4*** driver for SPI and Quad-SPI memories, the ***SerMem8*** driver for Octo-SPI memories and also the ***SerMem2x4*** driver for these particular Twin memories. This division is totally necessary to properly manage all the peculiarities of these memories and, especially, to exploit the full potential of each device: our mission is always to achieve the fastest flashing times.

As said before, the hardware connections are basically the same as an Octo-SPI memory. In the table below you can see more precisely how the external signals are connected to the internal memories:

Internal memory	Internal signal	External signal
Memory #1	MOSI	IO0
Memory #1	MISO	IO1
Memory #1	WP	IO2
Memory #1	HOLD	IO3
Memory #2	MOSI	IO4
Memory #2	MISO	IO5
Memory #2	WP	IO6
Memory #2	HOLD	IO7
Memory #1 and #2	CS	CS
Memory #1 and #2	CLK	CLK

It should appear clear how it is not possible to manage the two internal memories in parallel and independently since the CS and CLK lines are in common. For this reason, our driver manages them in parallel and synchronously, in other words, the same commands (with different parameters) are sent to both memories. Our solution is not so easy to manage but significantly improves the performances and it is actually the same solution used by the processors which typically use these memories in the customers’ applications.

The ***SerMem2x4*** driver allows the user to choose if using eight data lines (Twin-Quad-SPI), i. e. the typical Octo-SPI interface which is the fastest, or selecting a more robust protocol by only using the MOSI and MISO lines of each memory (Twin-SPI). This choice can be simply done using this parameter:

```
#TCSETPAR USE_8_DATA_LINES YES
```

One more complication about these twin memories concerns the division of the data between the two internal memories, since there is not a fixed standard, a certain freedom of use is left to the designers, who could make different choices depending on the application.
To better understand this problem, we have to make an example: let us consider a 512 Mbit twin memory (actually two 256 Mbit memories) and let us say that we want to write four bytes (0x12, 0x34, 0x56 and 0x78) from the address 0x0000_0000.
These are the possible configurations that could be used to split the data between the two memories:

1. Placing all data from the address 0x0000_0000 to 0x01FF_FFFF into the first memory and all the data from the address 0x0200_0000 to 0x03FF_FFFF into the second memory.
   So, all our four bytes (0x12, 0x34, 0x56 and 0x78) are placed at the beginning of the first memory.
   If you need to use this configuration, we can edit your source file to make it compatible with our driver.

2. Placing bytes at even addresses into the first memory and placing bytes at odd addresses into the second memory.
   So, 0x12 and 0x56 are placed at the beginning of the first memory, while 0x34 and 0x78 into the second memory.
   This is the configuration supported by default by our driver.

3. Placing bytes at odd addresses into the first memory and placing bytes at even addresses into the second memory.
   So, 0x34 and 0x78 are placed at the beginning of the first memory, while 0x12 and 0x56 into the second memory.
   If you need to use this configuration, you can simply swap the connection between memory #1 and memory #2.

4. Splitting bytes into two parts and placing the less significant part into the first memory and the most significant part into the second memory.
   So, 0x24 and 0x68 are placed at the beginning of the first memory, while 0x13 and 0x57 into the second memory.
   If you need to use this configuration, you can set the following parameter:

   ```
   #TCSETPAR SPLIT_BYTE YES
   ```

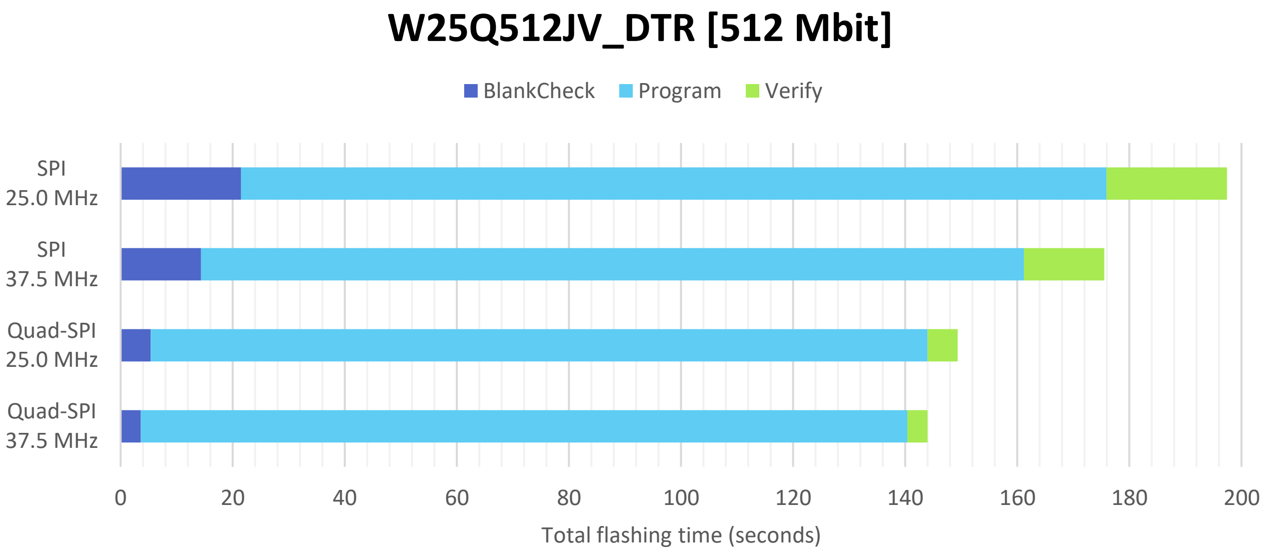
5. Splitting bytes into two parts and placing the most significant part into the first memory and the less significant part into the second memory.
   So, 0x13 and 0x57 are placed at the beginning of the first memory, while 0x24 and 0x68 into the second memory.
   If you need to use this configuration, you can use the configuration #4 and simply swap the connection between memory #1 and memory #2.

10. Flashing time examples

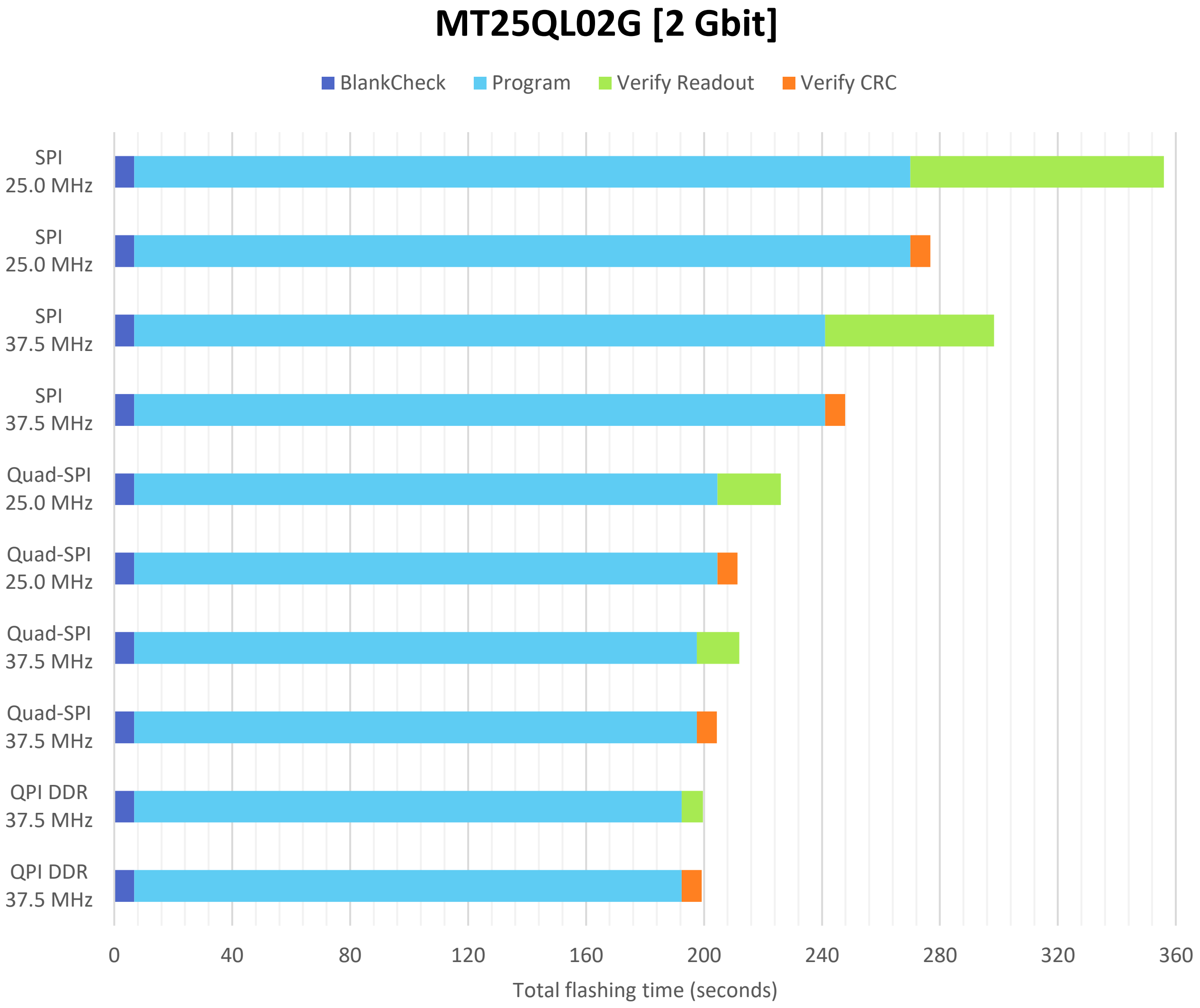
These are some examples of flashing times using our new drivers and OS > 3.00.

Additional notes:

1. Times were measured using FRB files containing random data to cover the entire memory of the target device.
2. We do not consider the masserase time in the total flashing time because it is usually skipped.
3. The time indicated for masserase is related to a virgin device.



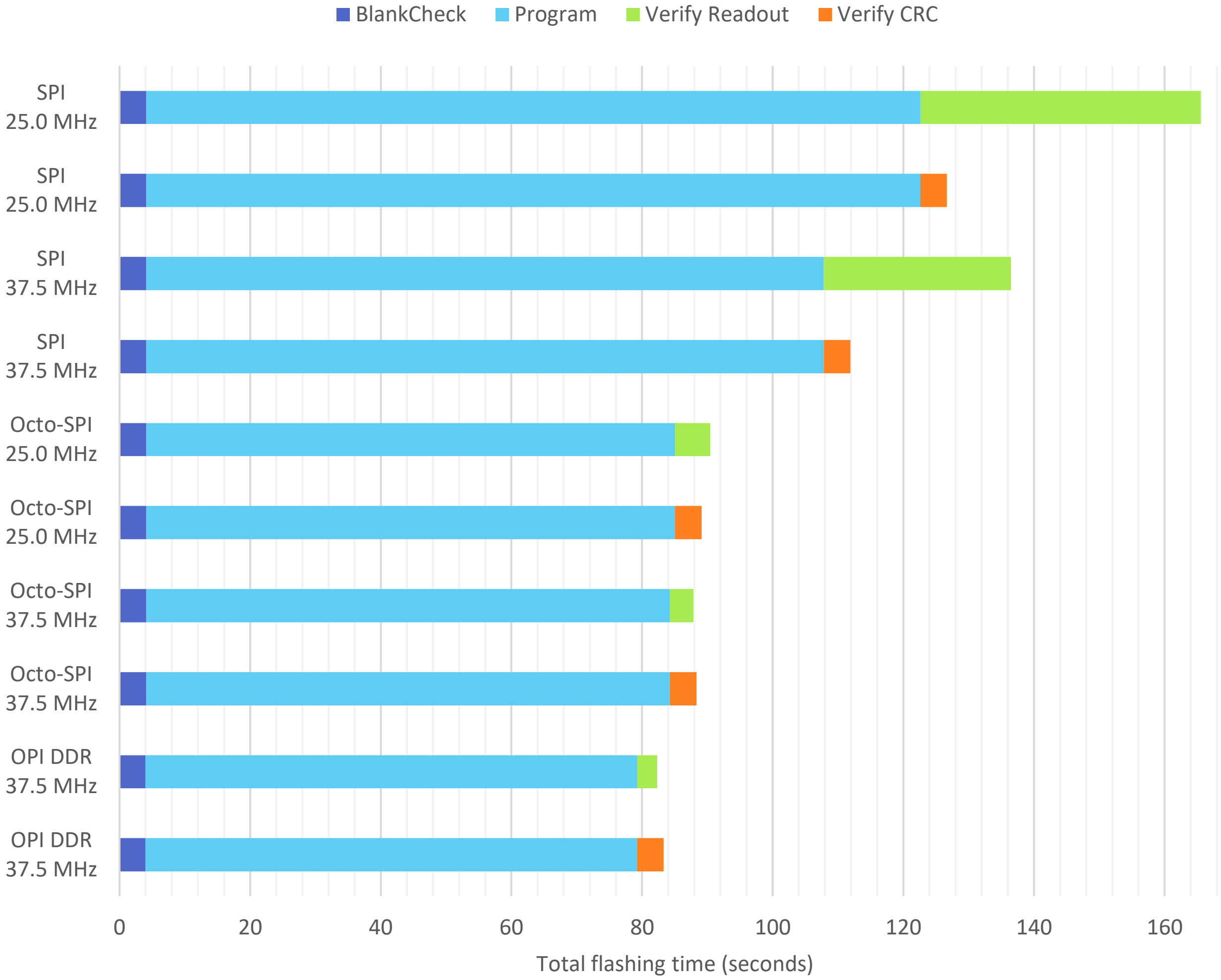
Protocol	Masserase	BlankCheck	Program	Verify
SPI 25.0 MHz	149.91 s	21.48 s	154.44 s	21.49 s
SPI 37.5 MHz	149.90 s	14.32 s	146.89 s	14.33 s
Quad-SPI 25.0 MHz	149.90 s	5.37 s	138.59 s	5.39 s
Quad-SPI 37.5 MHz	149.89 s	3.58 s	136.79 s	3.59 s



Protocol	Masserase	BlankCheck	Program	Verify	Verify CRC
SPI 25.0 MHz	3.35 s	6.83 s	263.14 s	85.95 s	6.83 s
SPI 37.5 MHz	3.32 s	6.82 s	234.23 s	57.31 s	6.83 s
Quad-SPI 25.0 MHz	3.33 s	6.82 s	197.70 s	21.53 s	6.82 s
Quad-SPI 37.5 MHz	3.31 s	6.81 s	190.78 s	14.36 s	6.82 s
Quad-SPI DDR 37.5 MHz	3.32 s	6.81 s	185.57 s	7.20 s	6.81 s

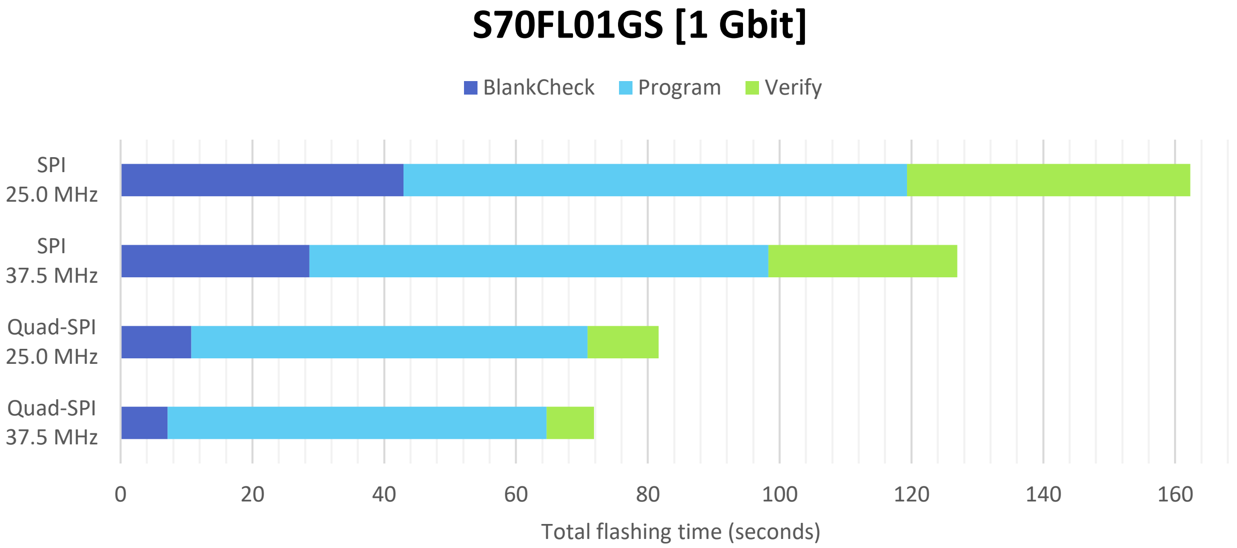
Note: blankcheck operation done using CRC.

MT35XU01G [1 Gbit]



Protocol	Masserase	BlankCheck	Program	Verify	Verify CRC
SPI 25.0 MHz	1.15 s	4.08 s	118.51 s	42.97 s	4.08 s
SPI 37.5 MHz	1.13 s	4.08 s	103.75 s	28.66 s	4.08 s
Octo-SPI 25.0 MHz	1.14 s	4.07 s	80.97 s	5.41 s	4.09 s
Octo-SPI 37.5 MHz	1.14 s	4.00 s	77.37 s	3.61 s	4.01 s
Octo-SPI DDR 37.5 MHz	1.13 s	3.99 s	75.29 s	3.05 s	4.01 s

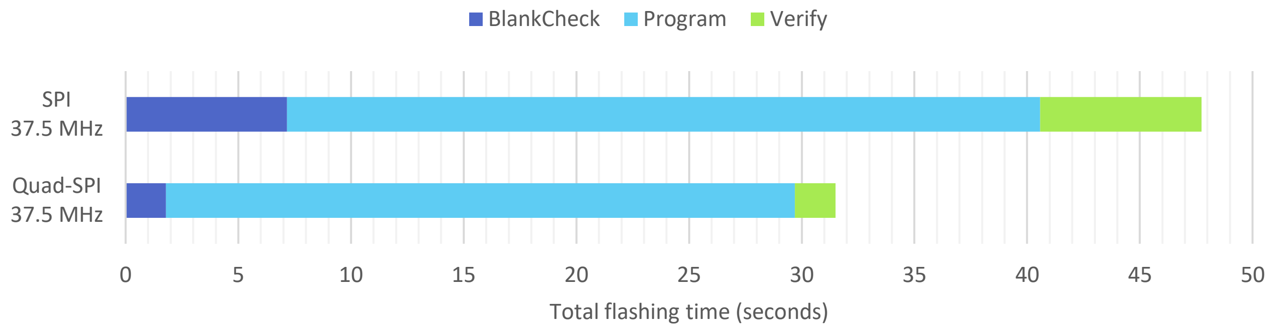
Note: blankcheck operation uses CRC.



Protocol	Masserase	BlankCheck	Program	Verify
SPI 25.0 MHz	114.83 s	42.95 s	76.40 s	42.97 s
SPI 37.5 MHz	112.23 s	28.63 s	69.65 s	28.66 s
Quad-SPI 25.0 MHz	111.06 s	10.74 s	60.12 s	10.77 s
Quad-SPI 37.5 MHz	114.40 s	7.16 s	57.50 s	7.19 s

Note: S70FL01GS is a device containing two S25FL512S sharing the same bus and they are managed as described in the chapter "Multiple memories on the same bus". Consider that a standard S25FL512S individually flashed takes **57.44 s** for the program operation in Quad-SPI at 37.5 MHz. This means that, basically, there are no differences between the programming time of the two serial memories inside the S70FL01GS and the programming time of a single S25FL512S memory.

IS25LP256D [256 Mbit]



Protocol	Masserase	BlankCheck	Program	Verify
SPI 37.5 MHz	63.81 s	7.16 s	33.42 s	7.16 s
Quad-SPI 37.5 MHz	62.46 s	1.79 s	27.91 s	1.80 s

Other flashing time examples

These are some flashing times of other devices. These times may result extremely useful because, as explained in chapter 3, the most relevant time is the one for the program and it depends mainly on the target device. This can be clearly seen in the table below.

Time for masserase is not reported because it is typically skipped for virgin devices.

Device	Size	Protocol	BlankCheck	Program	Verify	Total
XT25F32B	32 Mbit	Quad-SPI 37.5 MHz	0.22 s	3.99 s	0.22 s	4.43 s
MX25UW64	64 Mbit	Octo-SPI 37.5 MHz	0.22 s	4.99 s	0.22 s	5.43 s
AT25QF128A	128 Mbit	Quad-SPI 37.5 MHz	0.89 s	30.15 s	0.90 s	31.94 s
ATXP128	128 Mbit	Octo-SPI 37.5 MHz	0.45 s	304.84 s	0.45 s	305.74 s
IS25WP128	128 Mbit	Quad-SPI 37.5 MHz	0.89 s	13.75 s	0.90 s	16.78 s
MT25QL128	128 Mbit	SPI 37.5 MHz	0.42*s	13.04 s	0.42*s	13.88 s
MX25L12833F	128 Mbit	Quad-SPI 37.5 MHz	0.89 s	23.26 s	0.90 s	25.05 s

W25Q128JV	128 Mbit	Quad-SPI 37.5 MHz	0.89 s	24.28 s	0.90 s	26.07 s
IS25WX256	128 Mbit	Octo-SPI 37.5 MHz	0.89 s	17.29 s	0.90 s	19.08 s
GD25LB256E	256 Mbit	Quad-SPI 37.5 MHz	1.79 s	24.49 s	1.79 s	28.07 s
GD25LX256E	256 Mbit	Octo-SPI 37.5 MHz	0.89 s	23.38 s	0.90 s	25.17 s
MX25LM256	256 Mbit	Octo-SPI 37.5 MHz	0.89 s	19.87 s	0.90 s	21.66 s
MX66L51285G	512 Mbit	Twin-Quad 37.5 MHz	1.79 s	29.62 s	1.80 s	33.21 s
S25HL512T	512 Mbit	Quad-SPI 37.5 MHz	0.97*s	99.74 s	0.97*s	101.68 s
S25FL512S	512 Mbit	Quad-SPI 37.5 MHz	3.58 s	57.44 s	3.59 s	60.61 s
S25FS512S	512 Mbit	Quad-SPI 37.5 MHz	3.58 s	69.13 s	3.59 s	76.30 s
S79FL512S	512 Mbit	Twin-Quad 37.5 MHz	1.79 s	40.65 s	1.80 s	43.24 s
S28HS512T	512 Mbit	Octo-SPI 37.5 MHz	0.96*s	89.55 s	0.96*s	91.47 s
GD55LB01GE	1 Gbit	Quad-SPI 37.5 MHz	7.16 s	74.01 s	7.18 s	88.35 s
GD55LX01GE	1 Gbit	Octo-SPI 37.5 MHz	3.58 s	64.35 s	3.61 s	71.54 s
IS25LP01G	1 Gbit	Quad-SPI 37.5 MHz	7.16 s	150.75 s	7.19 s	165.10 s
MT25QU01G	1 Gbit	Quad-SPI 37.5 MHz	3.36*s	93.31 s	3.36*s	99.93 s
MT25TL01G	1 Gbit	Twin-Quad 37.5 MHz	1.73*s	48.33 s	1.73*s	51.79 s
S70FS01GS	1 Gbit	Quad-SPI 37.5 MHz	7.16 s	133.64 s	7.19 s	147.99 s
MT35XU512_x3	1.5 Gbit	Octo-SPI 30.0 MHz	2.12*s	40.79s	2.12*s	45.03 s

* Operation done using CRC.

11. Frequently Asked Questions

Some answers to the most frequently asked questions:

Are these features 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.

How many channels are needed to use Octo-SPI?

As mentioned before, Octo-SPI needs eight data lines, a clock signal, and the chip select line. This means ten DIOs of the FlashRunner and they cannot be managed by a single channel, so Octo-SPI needs a pair of channels, for example, channels 1 and 2, or 3 and 4, etc.
This implicates some additional limitations on the *FlashRunner NXG* and the *FlashRunner 2.0*: when using this special mode, the FlashRunner can parallelly flash only devices contained in the SerMem4 and SerMem8 drivers and the devices can only be placed starting from the odd channels.
For example, the user can use channel 1 to flash an IS25LP256D in Quad-SPI, while using channel 3 and 4 to flash an MT35XU02G in Octo-SPI and it is not possible to use channel 2 at the same time.
It is not even possible to flash devices which are not serial memories when using the FlashRunner in Octo-SPI mode. For example, in case the user also has to flash an Aurix TC397 using the same FlashRunner used to flash the two devices in the previous example, then the Aurix TC397 can be connected to channel 2 and it can be flashed before starting the operations on the two serial memories or after completing them, but not at the same time, not even using channel 1 instead of channel 2. In these situations, typically customers prefer to use two independent FlashRunner to use one in Octo-SPI mode and the other one in standard mode.
Instead, for the *FlashRunner HS*, the situation is easier because there are dedicated active modules which do the job, so more combinations are possible.

What is causing verify readout errors on Quad-SPI or Octo-SPI?

This is a common problem that happens when trying to use Quad-SPI or Octo-SPI with a non-optimal hardware setup. This error is caused by electromagnetic noise and it happens especially when the FlashRunner has to read a big amount of data from the target device. The very first thing to do to fix this problem is to use SPI, this should always work.
Instead, to keep using Quad-SPI and Octo-SPI, one possible solution is lowering the protocol frequency. If it is not working yet, then it means that the hardware setup requires a review so, please, see the chapter “Hardware setup”.

How to manage non-volatile registers used by FlashRunner?

On some devices which only have non-volatile registers to enable Quad-SPI, FlashRunner will set or reset the QE bit of the register according to the communication protocol requested. If the value set by FlashRunner does not fit the requirements of the customer, it is possible to simply program and verify those registers just before the end of the project. See the chapter “Register operations”.

How to manage alternative part numbers?

Sometimes customers have alternative part numbers from different silicon producers that can be mounted on the same board. So, it is possible that the FlashRunner has to flash a device chosen from a couple of alternatives. To understand which is the actual part number mounted on the board, customer can use the command **`READ_ID`**, which returns the JEDEC ID of the memory connected to the FlashRunner and, then, the response can be easily managed by the customer application.