SR5 E1 line: SDK EEPROM emulation driver

Introduction

External EEPROMs are often used in automotive applications to store adaptive/evolutive data. On the other hand, the microcontrollers used in those systems are more and more based on embedded-flash.

The trend to continuously reduce the number of components is forcing ECU designers to use flash memory to emulate EEPROM.

This application note explains the differences between external EEPROMs and embedded-flash and provides advises on how to substitute external EEPROM to emulated-EEPROM using the on-chip flash of SR5E1 devices.

Although the concept is easy to explain and implement “as is”, there are some embedded aspects that must be taken into account.

This application note is organized in 3 parts:

- description of the differences between external EEPROMs and embedded-flash
- general description of EEPROM emulation concept
- introduction to embedded application aspects

1 Embedded-flash and EEPROM

Before describing the proposed concept for EEPROM emulation, it is important to remember the main differences between the embedded-flash memory of a microcontroller and serial external EEPROMs. Those differences are summarized in the following table.

Table 1. Differences between embedded flash memory and EEPROM.

Feature	EEPROM	Emulated EEPROM from embedded-flash
Write time	Some ms random byte: from 5 to 10 ms page: equivalent to hundred µs / word (5 to 10 ms per page)	Some µs (ex: 50 µs per word)
Erase time	N.A.	Seconds. depends on sector size (ex: 1.5 s for big sectors)
Write method	Once started, is not CPU dependent; needs only proper supply.	Once started, is CPU dependent: a CPU reset stops the write process even if supply remains within specification.
Write access	Serial: hundred µs random word: 92 µs page: 22.5 µs/byte	Parallel: from 40 to 250 ns, depending on the flash memory used. Very few CPU cycles per word.

1.1 Difference in write access time

As flash has shorter write access time, critical parameters can be stored faster in the emulated EEPROM than in a serial external EEPROM, thereby improving the robustness of the system if the same safety concept is kept.

1.2 Difference in writing method

One of the important differences between external EEPROM and emulated EEPROM for embedded applications is the writing method.

- Stand-alone external EEPROM: once started by the CPU, the writing of a word cannot be interrupted by a CPU reset. Only supply failure will interrupt the writing process; so properly sizing the decoupling capacitors can secure the complete writing process inside a stand-alone EEPROM.
- Emulated EEPROM from an embedded-flash: once started by the CPU, the writing can be interrupted by a power failure and by a CPU reset.

This difference should be analyzed by system designers to understand the possible impact in their applications and to define the proper handling method.

1.3 Difference in erase time

The other important difference between stand-alone EEPROMs and emulated EEPROM with embedded-flash is the erase time. Unlike flash, EEPROM does not require a block erase operation to free-up space before write. This means that some form of software management is required to store data in flash. Moreover, as the erase process of a block in the flash takes few seconds, power shut-down and other spurious events that may interrupt the erase process (ex: reset) should be considered when designing the flash management software. This means that to design a robust flash management software it is necessary to have a deep understanding of the flash erase process.

1.4 Additional generic information on flash

SR5E1x includes a Data flash (4 × 16 KB blocks in one partition) that can be used for the EEPROM emulation. Data flash is programmable by word (32 bits) or doubleword (64 bits). ECC granularity is 64 bits.

The chosen ECC allows some bit manipulation so that a doubleword can be re-written several times without needing an erase of the block. This allows the use of a doubleword to store flags useful for the EEPROM emulation.

The sequence provided Table 2. Bits manipulation: doublewords preserving ECC integrity is related to data flash (therefore it would not apply on code flash). The ECC coding allows, starting from an all 1's doubleword value, to rewrite subsequent predefined code values changing only 1 s into 0 s and preserving the ECC integrity.

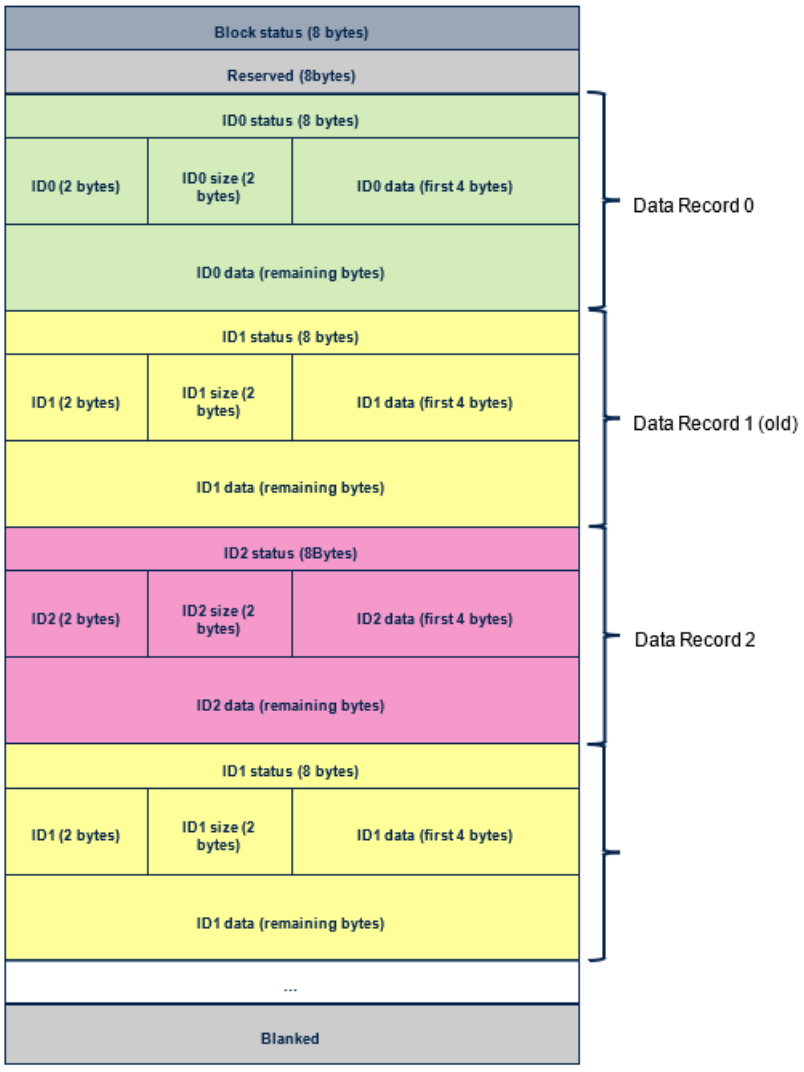
Table 2. Bits manipulation: doublewords preserving ECC integrity

Double word	ECC parity bits	Sequence
0xFFFF_FFFF_FFFF_FFFF	0x7FFF	Erase
0xFFFF_FFFF_FFFF_FFFE	0x3C88	Write1
0xFFFF_FFFF_FEFE_FC80	0x3C88	Write2
0xFFFF_FFFC_FEFC_FC80	0x1880	Write3
0xFCFC_FFFC_FCFC_F880	0x1880	Write4
0xFCF8_F8F0_FCE0_E080	0x1800	Write5
0xF8F8_E080_F0E0_E000	0x1000	Write6
0xE0E0_E080_E0E0_C000	0x1000	Write7
0xE000_8000_00C0_0000	0x1000	Write8

2 EEPROM emulation

The figure below shows the EEPROM emulation memory layout for the EEPROM active block.

Figure 1. EEPROM active block memory layout



Each EEPROM emulation block contains:

1. Block status (8 bytes). It can be one of the following values:
   a. ACTIVE
   b. INVALID
   c. SWAP

2. Data records. Each data record is composed by four fields:
   a. Data record status (8 bytes). It can be one of the following values:
      i. WRITE_START
      ii. WRITE_COMPLETE
      iii. INVALID
      iv. BAD
   b. Data record ID (2 bytes). It is the number that identifies the data record
   c. Data size (2 bytes). It is the size in bytes of the data record
   d. Data (4 bytes for data size <= 4 bytes, rounded to 8 bytes for data size > 4 bytes)
3. Blank space. It is the remaining available space in the data flash block for storing the new data records.

The current version of EEPROM emulation uses four blocks of the data flash: one block is the active block, while the other three can be used as spare block. Only one block at a time can be marked as ACTIVE. The ACTIVE block is the block currently in use that contains the valid data records. When no more space to write a new data record is available in the block B[i+1], the swap procedure is started: the block B[i+1] is marked as SWAP, then all valid data records are moved from B[i] in the spare block B[i+1]. When all data records are moved in the block B[i+1], the block B[i] will be marked as INVALID, while the spare block B[i+1] will become the new ACTIVE block.

The EEPROM emulation is designed so that each record fills a multiple of 8 bytes. When a new data record must be written, the free space needed to store the new data record is searched. If the free space needed is found, the first write sets the data record status to WRITE_START. Then the record ID, the record size and the data are written. When the writing of the data is completed, the data record status is marked as WRITE_COMPLETE. All records with a record status set to WRITE_COMPLETE are valid records. When a data record is updated or deleted, the data record status of the original record is set to INVALID. Moreover, in case of updating, a new record with the same ID will be written. All the records marked as INVALID will be skipped by the next swap procedure. When the EEPROM emulation is restarted, a scan of all the records available in the ACTIVE block will be executed. If the scan procedure founds a data record with the record status set to WRITE_START, the record will be marked as BAD. In fact, if a record has a record status set to WRITE_START, this means that a previous programming of ID, size or data has not been correctly completed. All the data records marked as BAD will be skipped from the next swap procedure.

Since the EEPROM emulation driver adopts variable data record lengths, each record will have a “size” field to identify the actual data size. The next data record location can be obtained by the current record start address and size.

When a record must be read, more records with the same ID (because of data updating) can be stored in the ACTIVE block, so the reading routine must identify the latest copy of data record by scanning the whole ACTIVE block for the first data record to the blank region. The valid instance of the record is the first one with the record status set to WRITE_COMPLETE. All the previous instances of the record are marked as INVALID.

3 Power loss management

Unmanaged power loss events could cause memory inconsistency states, resulting in read errors or data loss.

Power loss can still affect the data stored in EEPROM in the following ways:

- **incomplete write**: when data is being written to the EEPROM and power loss occurs, the write operation may not be complete resulting in incomplete or corrupted data.
- **incomplete swap**: when a swap operation is in progress and power loss occurs, the operation may not be complete resulting in incomplete or corrupted data.

3.1 Power loss during record writing operation

The write operation is structured in different steps, see the table below.

For simplicity in this document the writing steps are summarized, the first searching operation of a record with the same ID is not mentioned.

Table 3. Write steps of new record, with already existing ID

Write steps	Description	Power loss recovery	Record status
1	Write PROGRAM_START tag	NO (since data program has not fully completed, impossible to recover)	Only tag is memorized
2	Write ID record and its dimension	NO (since data program has not fully completed, impossible to recover it)	Only tag, ID and dimension are memorized
3	Write data with 8 bytes resolution	NO (since data program has not fully completed, impossible to recover it)	Undefined the state of the written data
4	Write PROGRAM_COMPLETE tag	YES	Invalidate the old record if it already exists
5	Identify old record and mark it as ID invalid	YES	No inconsistencies found

The write operation steps of new record are listed:

- **First step**: the PROGRAM_START tag is recorded in the first available position. The bytes that follow are still set to the default value of Flash memory since the ID and size fields will be written in the next step. Additionally, all operations are performed on the currently active block.
- **Second step:** the ID and size values are written simultaneously. ID equal to one already present has been chosen, to consider all cases.
- **Third step:** the user data is written. As previously described, the data is written in blocks of 8 bytes at a time, filling empty positions with default flash memory values. This step would be repeated several times to fully write the data.
- **Fourth step:** After fully programming the data need to change the record label from the value PROGRAM_START to the value PROGRAM_COMPLETE, which indicates that the record writing operation has been successfully completed. At this step, if a power loss occurred, there are two records with the same ID, both marked as PROGRAM_COMPLETE. Obviously, this is not a valid state, there are two valid data with the same ID.
- **Fifth step:** The last operation is to mark the previous record as invalid by setting the tag to ID_INVALID. Obviously, this step is performed if and only if the data is to be updated, i.e., if there is a record with the same ID.

In case of power loss, for the steps from **1** to **3**, the driver marks the record as damaged, and the data are not recoverable. This means that, during the initial scan, if a record with PROGRAM_START tag is found, it will be set to the value ID_BAD.

In case of power loss, for the steps from **4** to **5,** after writing the data, the driver changes the status of the new record to PROGRAM_COMPLETE. Although unlikely, a power loss after this moment and before the invalidation of the previous record must be managed to ensure the driver is safe from power loss.

If the last record status is not PROGRAM_COMPLETE, there was no power loss during the deletion step of the previous record. No additional action is needed.

If the status of the last read record is PROGRAM_COMPLETE, the search function is invoked on the ID of the last record. It returns the first address occurrence of the desired record. The status of this last search ID is reprogrammed with ID_INVALID tag.

3.2 Power loss during record swapping operation

The swap operation is structured in different steps, see the table below.

Table 4. Block status during swap procedure

Swap steps	Description	Active block	Spare block	Power loss recovery
1	Spare block is marked as SWAP	ACTIVE	SWAP	YES
2	Data is swapped	ACTIVE	SWAP	YES
3	Active block is marked as INVALID	INVALID	SWAP	YES
4	Spare block is marked as ACTIVE	INVALID	ACTIVE	YES

The swap operation steps are listed:

- **First step:** The active block is full, so it is needed to individuate the new block. Check its status and eventually delete it. The new status is marked as SWAP_START.
- **Second step:** data is copied in the new block, marked as SWAP_START.
- **Third step:** the ACTIVE block is marked as INVALID.
- **Fourth step:** the new block is marked as ACTIVE.

In case of power loss for the steps from **1** to **2,** the swap operation is restarted**.** The data integrity is not compromised, an active block is always present during these steps.

Although highly unlikely, a power loss during the steps from **3** to **4** could result in the full data loss if power loss management is not implemented**.**

In this condition both blocks contain valid data, the new block will be marked as ACTIVE to avoid the repetition of the swap operation.

4 EEPROM APIs

4.1 eed_init

```
void eed_init(eed_driver_t *edp)
```

Table 5. eed_init parameters.

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure

This API initializes the EEPROM emulation. It must be invoked before any other API.

4.2 eed_start

```
uint32_t eed_start(eed_driver_t *edp)
```

Table 6. eed_start parameters

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure
OUT	operation status	EED_OK if no error occurs, otherwise one of the following error codes: EED_ERROR_FLASH_INIT EED_ERROR_FLASH_UNLOCK EED_ERROR_FLASH_ERASE EED_ERROR_FLASH_PROGRAM EED_ERROR_SWAP EED_ERROR_ID_NOT_FOUND

This API starts the EEPROM emulation. It unlocks all the blocks used by the EEPROM emulation. Then, it searches for an ACTIVE block. If no ACTIVE block is found, all blocks used by the EEPROM emulation will be erased, then the first one will be marked as ACTIVE. Instead, if an ACTIVE block is found, all the data records available in the block will be scanned to establish the free space available in the block.

Table 6. eed_start parameters shows the error codes related to this API.

Table 7. eed_start error codes

Error code	Description
EED_ERROR_FLASH_INIT	It occurs if the data flash driver initialization fails
EED_ERROR_FLASH_UNLOCK	It occurs if the data flash driver is not able to unlock the data flash
EED_ERROR_FLASH_ERASE	It occurs if the data flash driver is not able to erase the data flash
EED_ERROR_FLASH_PROGRAM	It occurs if the data flash driver is not able to program the data flash
EED_ERROR_SWAP	It occurs if the swap procedure is not correctly completed
EED_ERROR_ID_NOT_FOUND	It occurs if the data record is not found

4.3 eed_write_id

```
uint32_t eed_write_id(eed_driver_t *edp, uint16_t id, uint16_t size, uint32_t
source)
```

Table 8. **eed_write_id parameters**

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure
IN	id	identifier of the data record to be written
IN	size	size of the data record to be written
IN	source	Address of the buffer containing the data to be written
OUT	operation status	EED_OK if no error occurs, otherwise one of the following error codes: EED_ERROR_FLASH_PROGRAM EED_ERROR_NOT_INITIALIZED EED_ERROR_NO_SPACE EED_ERROR_SWAP

This API writes a data record. It searches for the record ID. If a data record with the same ID is already present, the new version will be written, then the previous one will be invalidated. If no more space available in the ACTIVE block is found, a swap procedure is started to free space to store the new data record. This API can be invoked only after the `eed_start`.

Table 8. eed_write_id parameters shows the error codes related to API.

Table 9. **eed_write_id error codes**

Error code	Description
EED_ERROR_FLASH_PROGRAM	It occurs if the data flash driver is not able to program the Data flash
EED_ERROR_NOT_INITIALIZED	It occurs if the EEPROM emulation is not initialized
EED_ERROR_NO_SPACE	It occurs if no more space is available in the EEPROM emulation
EED_ERROR_SWAP	It occurs if the swap procedure is not correctly completed

4.4 eed_read_id

```
uint32_t eed_read_id(eed_driver_t *edp, uint16_t id, uint16_t *size, uint32_t source)
```

Table 10. **eed_read_id parameters**

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure
IN	id	identifier of the data record to be read
OUT	size	size of the data record to be read
OUT	source	Address of buffer in which read data is stored
OUT	operation status	EED_OK if no error occurs, otherwise one of the following error codes: EED_ERROR_NOT_INITIALIZED EED_ERROR_ID_NOT_FOUND

This API reads a data record. It searches for the record ID. This API can be invoked only after the `eed_start`. The table below shows the error codes related to this API can return.

Table 11. **eed_read_id error codes**

Error code	Description
EED_ERROR_NOT_INITIALIZED	It occurs if the EEPROM emulation is not initialized

Error code	Description
EED_ERROR_ID_NOT_FOUND	It occurs if the data record is not found

4.5 eed_delete_id

`uint32_t eed_delete_id(eed_driver_t *edp, uint16_t id)`

Table 12. eed_delete_id parameters

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure
IN	id	identifier of the data record to be deleted
OUT	operation status	EED_OK if no error occurs, otherwise one of the following error codes: EED_ERROR_FLASH_PROGRAM EED_ERROR_NOT_INITIALIZED EED_ERROR_ID_NOT_FOUND

This API deletes a data record. It searches for the record ID. The data records will not be physically deleted from the data flash, but it will be invalidated (its status will be marked as INVALID). This API can be invoked only after the `eed_start`.

The table below shows the error codes related to API.

Table 13. eed_write_id error codes

Error code	Description
EED_ERROR_FLASH_PROGRAM	It occurs if the data flash driver is not able to program the data flash
EED_ERROR_NOT_INITIALIZED	It occurs if the EEPROM emulation is not initialized
EED_ERROR_ID_NOT_FOUND	It occurs if the data record is not found

4.6 eed_get_freespace

`uint32_t eed_get_freespace(eed_driver_t *edp)`

Table 14. eed_get_freespace parameters

Parameters		Description
IN	edp	Pointer to a @p eed_driver_t structure
OUT	freespace	Free space [bytes] available

This API returns the free space (in bytes) already available in the EEPROM emulation. This API can be invoked only after the `eed_start`.

4.7 eed_remove

`uint32_t eed_remove(eed_driver_t *edp)`

Table 15. eed_remove parameters

Parameters		Description
IN	edp	pointer to a @p eed_driver_t structure
OUT	operation status	EED_OK if no error occurs, otherwise one of the following error codes: EED_ERROR_FLASH_ERASE

Parameters		Description
		EED_ERROR_NOT_INITIALIZED

This API removes the EEPROM emulation. It deletes the whole contents of the EEPROM emulation. All data records will be lost. This API can be invoked only after the `eed_start`.

The table below shows the error codes related to what this API can return.

Table 16. eed_remove error codes

Error code	Description
EED_ERROR_FLASH_ERASE	It occurs if the Data flash driver is not able to erase the Data Flash
EED_ERROR_NOT_INITIALIZED	It occurs if the EEPROM emulation is not initialized

Revision history

Table 17. **Document revision history**

Date	Revision	Changes
24-Jul-2023	1	Initial release.
09-Jan-2024	2	Updated Figure 1. EEPROM active block memory layout and Section 2 EEPROM emulation; Table 6. eed_start parameters and Table 7. eed_start error codes; deleted section **eed_set_cfg**; added Section 3.1 Power loss during record writing operation and Section 3.2 Power loss during record swapping operation. Minor text changes.

Contents

List of tables