Technical note

SPC58x configuring CAN and CAN-FD filters

Introduction

This technical note describes how to configure acceptance filters for CAN controllers embedded in SPC58x automotive microcontrollers. The document describes the register configurations and provides some example to speed up the filter configuration. These configurations can be adopted for all microcontrollers in this family with minor variances.

The examples in this document are based on SPC584Cx/SPC58ECx 32-bit MCU.

1 CAN overview

SPC584Cx/SPC58ECx has eight CAN instances embedded in two different subsystems as documented in the device reference manual Section Appendix A Reference documents.

All the CAN controllers in the same subsystem will share resources like RAM memory, clock, etc.

Each CAN subsystem is constituded by the following major blocks:

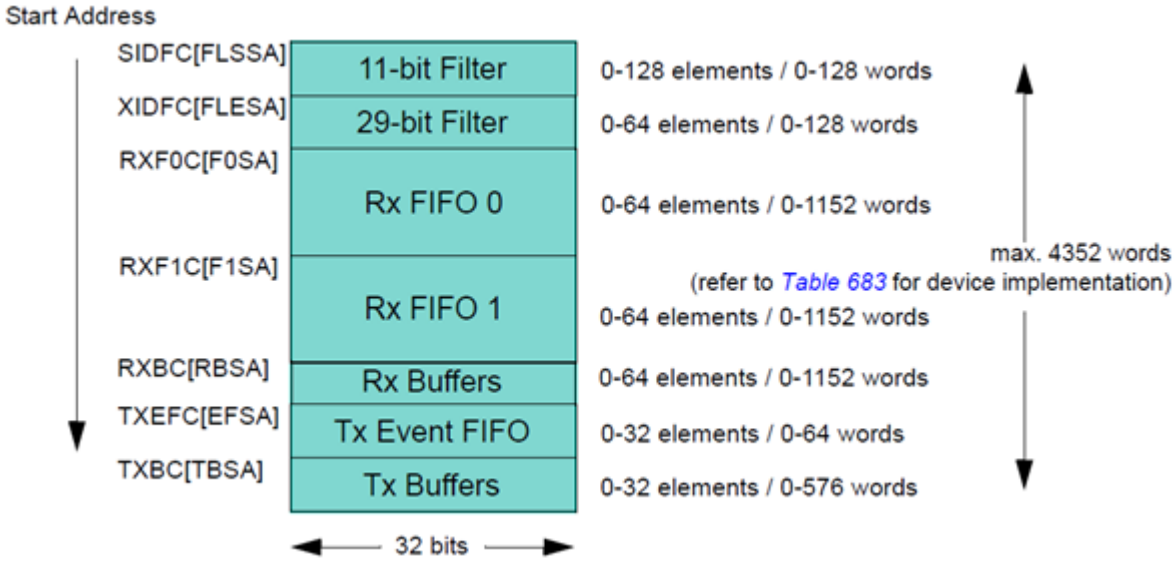
- Modular CAN cores: the registers of the CAN module can be accessed using the Generic Slave Interface (GSI). The peripheral GSI module acts as a request from each master.
- CAN-RAM arbiter: it is an additional logic for arbitration among the requests for the RAM access by the various CAN controllers.
- SRAM: the CAN subsystem will interface with an external RAM using this interface, it is the SRAM.
- ECC controller: it contains the logic to compute and validate the correction code on the SRAM memory.

For the SRAM interface and memory organization refer to the device reference manual Section Appendix A Reference documents.

2 Filtering introduction

CAN filter logic allows you to configure the filters in various way. For example, the messages that pass the acceptance filtering can be stored in Rx FIFO (0 or 1) or in dedicated rx buffers. Each filter can be configured as an **acceptance** or **rejection** filter and can also be enabled or disabled. For acceptance filter, each filter list is performed from item #0 to the first matching item in the filter list. Before using filters it is mandatory to configure the starting address and the numbers of the filter to reserve part of message RAM. The figure below shows a view of the shared memory map and the registers (hence the starting addresses for each section).

Figure 1. Message RAM configuration example



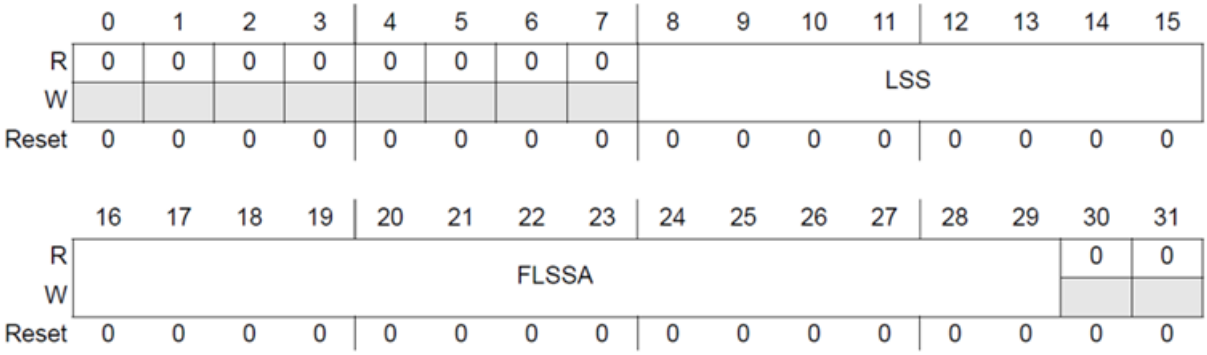
3 Message RAM initialization

Before using any filter, it is mandatory to configure the relevant RAM area of the messages in which they will be stored. To do this, the software application must write the offset (in words) from the message RAM base address of each filter area. The number of filters for each area must be configured, so that the controller can understand where each related filter memory area ends

In this document, as example four filters, will be configured for standard identifiers (11 bits) and four filters for extended identifiers (29 bits), so, reserve a portion of message RAM to store 11-bit filters and another for 29-bit filters. To configure the starting address of the standard id filter area, the software must write the FLSSA field of the SIDFC register (Standard ID filter configuration register).

For extended ID filters it is necessary to write the FLESA field of the XIDFC register (Extended ID filter configuration register). The FLSSA and FLESA fields should contain the memory offset "in words" from the RAM base address of the message. This application configures four standard filters at offset zero and four extended filters.

Figure 2. SIDFC register



For standard filter configuration:

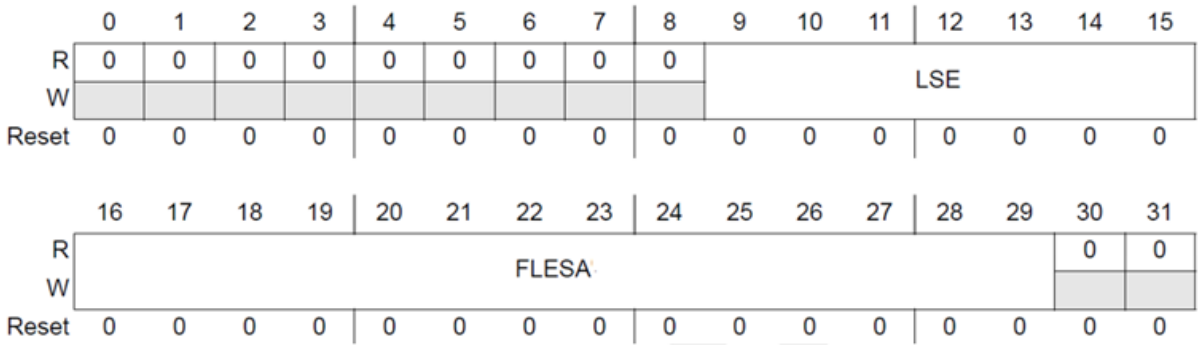
- FLSSA = 0x0: the offset respect to message RAM base address is zero, so the area starts at the beginning of the Message RAM.
- LSS = 4: this is the number of filters to configure. Each filter is composed by `one` 32 bits word.

In this configuration a portion of memory starting from offset zero and having a size of four words.

Note: the CAN controller has no control mechanism for configuring the message RAM, this means that the Developer must be careful not to overlap the configured RAM areas.

The following figure shows how to configure a portion of memory to store extended identification filters.

Figure 3. XIDFC register



To configure the RAM area to store extended filters it is necessary to program the following values in the FLESA and LSE fields.

- FLESA = 0x04: the offset in words respect to message RAM base address. This because in previous configured filters area it has been reserved 0x04 words for standard ID filters and then the minimum usable offset is 0x04.
- LSE = 4: this is the number of filters to configure. For extended ID a filter is composed by `two` 32 bits word.

In this configuration a portion of memory starting from offset 0x04 (words) having a size of eight words (four two-word filters). Therefore, the minimum offset for the next configurable memory area is 0x0C words.

All sections of the message RAM must be configured considering the number and size of the elements that will be stored in the section without overlapping any sections.

Note: to convert a word offset into a byte offset, it is needed to multiply the word value by four.

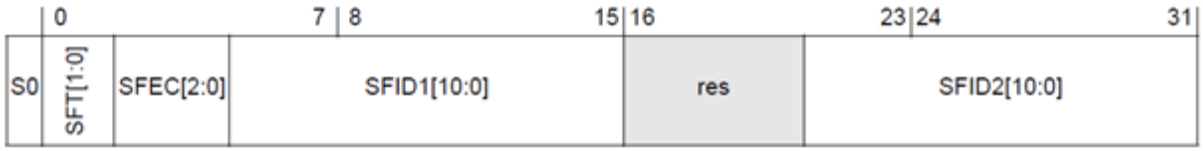
4 Example of standard ID filters

After configuring the message RAM, you can configure the device filters.
Each filter element can be configured as:

- Range filter
- Double ID filter
- Classic filter
- Filter for dedicated rx buffer (single ID filter)

The following figure shows how to configure four different types of filters for standard identifiers (11-bit identifiers).
Use the following filter element register for STANDARD ID.

Figure 4. Filter element register



4.1 Range filter for RX FIFO0

Below an example of filtering to store the messages with identifiers in the range [0x16 , 0xF6] into the Receive FIFO 0.
Register value (HEX): `0x081600F6`
Register value (BIN): `00 001 00000010110 (00000) 00011110110`

Note: in all the examples provided in this document, the bracketed bits are kept at the default value.

Fields values:

```
SFT -> `00` -> Range filter from SFID1 to SFID2
```

```
SFEC -> `001` -> Store in Rx FIFO 0 if filter matches
```

```
SFID1 -> `00000010110` -> First ID of standard ID filter element range (0x16)
```

```
SFID2 -> `00011110110` -> Second ID of standard ID filter element range (0xF6)
```

4.2 Double filter for RX FIFO1

Below is an example of a double ID filter to store messages identifiers 0 x 0A or 0 x FF into FIFO 1.
Register value (HEX): `0x500A00FF`
Register value (BIN): `01 010 00000001010 (00000) 00011111111`
Fields values:

```
SFT -> `01` -> Dual ID filter for SFID1 or SFID2
```

```
SFEC -> `010` -> Store in Rx FIFO 1 if filter matches
```

```
SFID1 -> `00000001010` -> First ID of standard dual ID filter element
```

```
SFID2 -> `00011111111` -> Second ID of standard dual ID filter element
```

4.3 Rx dedicated buffer filtering

Below is an example to store messages with identification 0 x 7F0 into dedicated rx buffer #0.

Using a dedicated buffer, it is possible to filter only one message ID and that is the one written in the SFID1 field.

Register value (HEX): `0x3FF00000`

Register value (BIN): `00 111 11111110000 (00000) 00 (000) 000000`

```
SFT -> `00` -> this value isn`t important because filtering to store into dedicated RX
BUFFER and in this case SFT value is
        ignored (see SFEC field description case `111`)
```

```
SFEC -> `111` -> Store into dedicated rx buffer if filter matches
```

```
SFID1 -> `11111110000` -> standard ID that filter will accept (0x7F0)
```

`SFID2[10, 9] -> `00`` -> this field decides whether the received message is stored in an rx buffer or treated as message A, B, or C of the debug message sequence. In this case, the software application wants to store the message in an rx buffer.

`SFID2[0,5] -> `000000`` → index of the dedicated buffer rx where the corresponding message will be stored (if N dedicated buffer rx has been configured, this index can be in the range [0, N -1]).

In this case the message is stored into dedicated buffer #0.

4.4 Classic filter for RX FIFO0

This example shows how to store the messages with identifier in the range [0x688, 0x68F] in the RX FIFO 0. For a classic filter, an Identifier / Mask pair must be defined. The identifier must be written in SFID1 and the mask in SFID2 fields of filter element. In a classic filter, the IDs messages that pass the filter are obtained by applying the mask to the identifier as follows:

- this is the meaning of the filter bits -> 1 = must match (0 = don`t care);
- having a filter with a mask made up of all "1s", only one identifier will pass to the filter (the one that is written in the SFID1 field) while a filter having a mask made up of all "0" all identifiers will pass the filter.

Obviously, these are the simplest filters. Below is an explanation of how to organize a range filter for the standard ID in range [0x688, 0x68F] which stores received messages in RX FIFO 0.

This is a standard message filter, so we will give values to SFID1 = Identifier and SFID2 = Mask.

Register value (HEX): `0x8E8B07F8`

Register value (BIN): `10 001 11010001011 00000 11111111000`

```
SFT -> `10` -> Classic filter: SFID1 =
        filter, SFID2 = mask
```

```
SFEC -> `001` -> Store in Rx FIFO 0 if
        filter matches
```

```
SFID1 = 110 1000 1011 (Identifier
        0x68B)
```

```
SFID2 = 111 1111 1000 (Mask
        0x7F8)
```

Applying the mask to the identifier (a little bit) with the meaning of the mask bit (1 = must match 0 = don`t care) we find the following range filter where the `X` symbol stands for zero or one.

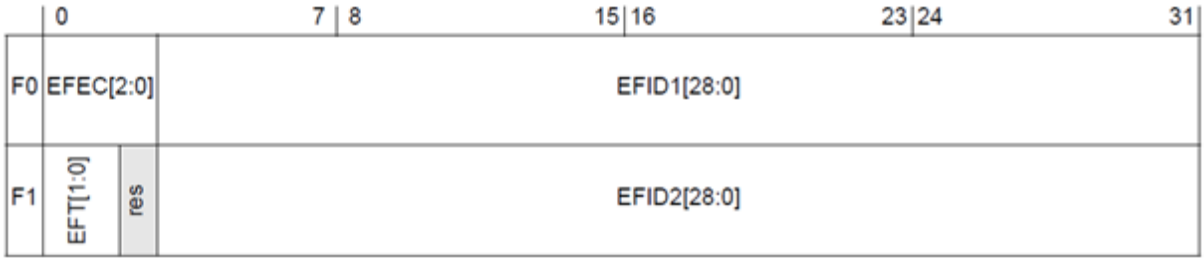
```
Filter =   110 1000
      1XXX
```

Then, all standard messages in the range [0x688, 0x68F] will pass the filter.

5 Extended ID filters configuration

The same filters types can be implemented also for extended identifiers (29-bit identifiers).

Figure 5. Extended filter element register



5.1 Range extended filter for RX FIFO0

In this example, this filter will store messages with identifiers in the range [0xFFFFF, 0x1FFFFFFF] in FIFO 0.
Below the register values:

F0 register value (HEX): `0x200FFFFF`

F1 register value (HEX): `0xDFFFFFFF`

FO register value (BIN): `001 00000000011111111111111111111`

F1 register value (BIN): `11 (0) 11111111111111111111111111111`

Fields values:

```
EFEC -> `001` -> Store in Rx FIFO 0 if filter matches
```

```
EFID1 ->
        `00000000011111111111111111111` -> First ID of extended ID filter element
range(0xFFFFF)
```

```
EFT -> `11` -> Range filter from SFID1 to SFID2
```

```
SFID2 ->`11111111111111111111111111111` -> Second ID of standard ID filter element range
        (0x1FFFFFFF)
```

5.2 Double ID filter for FIFO 1

In this example the dual ID filter will store messages having identifiers 0xAAAAA or 0xBBBBB into FIFO 1.

F0 register value (HEX): `0x400AAAAA`

F1 register value (HEX): `0x400BBBBB`

FO register value (BIN): `010 000000000 10101010101010101010`

F1 register value (BIN): `01 (0) 00000000010111011101110111011`

Fields values:

```
EFEC -> `010` -> Store in Rx FIFO 1 if  filter matches
```

```
EFID1 -> `00000000010101010101010101010` -> First extended ID (0xAAAAA)
```

```
EFT -> `01` -> Dual ID filter for EFID1 or EFID2
```

```
EFID2 -> `00000000010111011101110111011` -> Second extended ID (0x000BBBBB)
```

5.3 Dedicated rx buffer

In this example the filtering will store messages having identifier 0x000AAAAA into dedicated rx buffer #1. Also in this scenario, using the dedicated buffer only one message id can be filtered and it`s the one wrote into EFID1 field.

ID Filter for dedicated rx buffer (ID = 0x000AAAAA)

F0 register value (HEX): `0xE00AAAAA`

F1 register value (HEX): `0x00000001`

FO register value (BIN):

```
111 00000000010101010101010101010
```

F1 register value (BIN): `00 (0) 00000000000000000 00 (000) 000001`

Fields values:

```
EFEC -> `111` -> Store into dedicated rx  buffer if filter matches
```

```
EFID1 ->  `00000000010101010101010101010` -> extended ID that filter will accept
        (0x000AAAAA)
```

```
EFT -> `00` -> this value isn`t important because filtering to store into dedicated RX
BUFFER and in this case EFT value is
        ignored (see EFEC field description case `111`)
```

```
EFID2[10, 9] -> `00` -> this field decides whether the received message is stored into an Rx
Buffer or treated as message A, B,
        or C of the debug message sequence. In this case software application wants to store
message into an Rx Buffer
```

```
EFID2[0,5] -> `000001` -> index of the dedicated rx buffer where the matching message will
be stored (if you configured N dedicated
        rx buffer this index can be in the range [0, N -1]). In this case the message is
stored into dedicated buffer #1
```

5.4 Classic filter for rx FIFO1

In this example, the filter programming will store messages store with identifier in the range [0 x FFFFF, 0 x 1FFFFFF] into the rx FIFO 1. It is underatood that the masking method is the same as the standard ID filters.

Below is an explanation on how to program an extended ID range filter in rage [0 x FFFFF, 0 x 1FFFFFF] which stores received messages in rx FIFO 1. This is a filter for extended messages, so we will give values to EFID1 = Identifier and EFID2 = Mask

F0 register value (HEX): `0x400FFFFF`

F1 register value (HEX): `0x9E0FFFFF`

F0 register value (BIN): `010 00000000011111111111111111111`

F1 register value (BIN): `10 (0) 11110000011111111111111111111`

```
EFT > `10` -> Classic filter: EFID1 =  filter, EFID2 = mask
```

```
EFEC -> `010` -> Store in Rx FIFO 1 if filter matches
```

```
EFID1 = 0 0000 0000 1111 1111 1111 1111 1111     (Identifier 0xFFFFF)
```

```
EFID2 = 1 1110 0000 1111 1111 1111  1111 1111  (Mask 0x1E0FFFFF)
```

Applying the mask to the identifier (a bit) with the meaning of the mask bit (1 = must correspond to 0 = don`t care) we find the following range filter where the symbol `X` stands for zero or one.

```
Filter =    0 000X XXXX 1111 1111 1111 1111 1111
```

So, all extended messages in the range [0xFFFFF, 0x1FFFFFF] will pass the filter.

Appendix A Reference documents

- SPC584Cx/SPC58ECx Reference manual
- SPC584Cx/SPC58ECx datasheet

Appendix B Acronyms and abbreviations

Table 1. **Acronyms**

Abbreviation	Complete name
CAN	Controller area network
FD	Flexible data rate

Revision history

Table 2. **Document revision history**

Date	Version	Changes
01-Mar-2021	1	Initial release.

Contents

List of figures