FRONTGRADE

APPLICATION NOTE

UT32M0R500

CAN Filtering and its Application

Arm Cortex M0+

1/3/2021

Version #: 1.0.1

Product Name	Manufacturer Part Number	Smd #	Device Type	Internal Pic Number
Arm Cortex M0+	UT32M0R500	5962-17212	CAN Unit	QS30

Table 1: Cross Reference of Applicable Products

1.0 Overview

Unlike Ethernet and I^2C, CAN messages are "address-less", meaning that the messages do not contain destination address information. Any device attached to the CAN bus is capable of receiving any message placed on the bus. To prevent a device from being overwhelmed by unwanted messages, CAN peripherals can be programmed to perform message filtering. CAN filtering allows for the acceptance of specific messages based on the message ID, or to accept a range of messages based on a filter mask. CAN filters and masks direct the CAN peripheral to examine incoming messages and accept/reject them based on their contents. This document details the UT32M0R500 CAN filtering and how Frontgrade-supplied APIs map programmer-friendly filter/mask values into CAN.

2.0 UT32M0R500 CAN Peripheral

The CAN peripheral within Wolverine is based on the Philips SJA1000 CAN device, with a nearly identical register set. The CAN peripheral can operate in two modes: BasiCAN and PeliCAN. BasiCAN – as its name implies – is a very basic CAN peripheral that supports only standard (11-bit) ID frames and a single filter/mask pair. PeliCAN – which occupies the same register space as BasiCAN – supports both standard and extended (29-bit) ID frames, as well as enhanced filtering logic. The following table describes the differences in capabilities between modes. *[It is assumed that the reader is familiar with CAN frame structures and field definitions, including the ID bits, RTR bit, and data (payload) bytes…]*

Mode	Filter	Frame	Filter #1 ID bits tested	Filter #1 RTR bit tested	Data[0] byte tested	Data[1] byte tested	Filter #2 ID bits tested	Filter #2 RTR bit tested
BasiCAN	Single	Standard	10…3	n/a	n/a	n/a	n/a	n/a
PeliCAN	Single	Standard	10…3	TESTED	TESTED	Ignored	n/a	n/a
PeliCAN	Single	Extended	28…0	TESTED	Ignored	Ignored	n/a	n/a
PeliCAN	Double	Standard	10…3	TESTED	TESTED[1]	Ignored	10…3	TESTED
PeliCAN	Double	Extended	28…0	Ignored	Ignored	Ignored	28…0	Ignored

3.0 BasiCAN Filtering

In its most simple form – using BasiCAN – the filtering is best described as such:

Example #1

Assume that our application is interested in receiving only those CAN messages that have the (binary) ID of 10101101100_2 (or in hexadecimal, 0x56C). As indicated in the table above, the BasiCAN filter can be programmed for ID bits 10..3 only, which means we cannot filter on bits 2..0, indicated by 'XXX' below. With this criterion, the BasiCAN filter register is programmed as such:

```
BASICAN->ACCEPT_CODE = 10101101100b >> 3; // 101011012 after shift
```

Because only ID bits 10..3 are used for filtering, any message ID of $10101101XXX_2$ will pass through the CAN's filter. This means that our application will receive messages with IDs of 10101101000_2 through 10101101111_2. In this case, the application will need to perform a "second level" (software) filtering of all incoming messages to discard any message without an ID of 10101101100_2.

Example #2

To expand upon Example #1, let's assume our application now wants to receive **any** CAN messages where the upper six bits are 101011_2 or $101011yyXXX_2$. Due to the filter already ignoring ID bits 2..0, we need only focus on ID bits 4..3, indicated by 'yy' above. To get the CAN to ignore these two bits, we have the application program the mask register with 00000011000_2, which will appear in software as:

```
BASICAN->ACCEPT_MASK = 00000011000b >> 3; // 00000011₂ after shift
```

This will instruct the CAN peripheral to accept any messages with IDs from 10101100000_2 to 10101111111_2. In short, for every bit in the mask register that is programmed as a '1', the corresponding bit in the acceptance filter register is ignored. *[This implies that if the mask register is programmed with **all** 1s (0xFF), all CAN messages will pass through the filter!]*

4.0 PeliCAN Filtering

As alluded to in the table above, the PeliCAN filter set is substantially more flexible – and correspondingly more complicated – than what is provided with BasiCAN. There are four filter options available in PeliCAN mode:

- Single-Filter, Standard-Frame
- Single-Filter, Extended-Frame
- Dual-Filter, Standard-Frame
- Dual-Filter, Extended-Frame

These four filter options allow for a wider – and in some cases, narrower – array of "acceptable" messages from the CAN bus. We will address these options separately in a moment.

For PeliCAN, there are four acceptance filter registers and four mask registers. As with BasiCAN, for every bit in the (4-byte) acceptance filter register array, there is a corresponding bit in the (4-byte) mask array. **Again, any bit in the mask that is set to '1' instructs the CAN peripheral to ignore the corresponding bit in the acceptance filter.**

Single-Filter, Standard-Frame

This option is closest to the filter mechanism offered by BasiCAN but supports additional filter tests. The differences are:

- All 11 bits of the **standard** message ID can be tested
- The RTR bit can be tested
- The values of Data[1..0] (payload) can be tested

For the "Single-Filter, Standard-Frame" option, the four acceptance filter registers assume the following format:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID_10	ID_9	ID_8	ID_7	ID_6	ID_5	ID_4	ID_3
1	ID_2	ID_1	ID_0	RTR	n/a	n/a	n/a	n/a
2	Data[0].7	Data[0].6	Data[0].5	Data[0].4	Data[0].3	Data[0].2	Data[0].1	Data[0].0
3	Data[1].7	Data[1].6	Data[1].5	Data[1].4	Data[1].3	Data[1].2	Data[1].1	Data[1].0

And the four mask registers:

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID_10	ID_9	ID_8	ID_7	ID_6	ID_5	ID_4	ID_3
1	ID_2	ID_1	ID_0	RTR	n/a	n/a	n/a	n/a
2	Data[0].7	Data[0].6	Data[0].5	Data[0].4	Data[0].3	Data[0].2	Data[0].1	Data[0].0
3	Data[1].7	Data[1].6	Data[1].5	Data[1].4	Data[1].3	Data[1].2	Data[1].1	Data[1].0

Example #3

If we want our application to process only those CAN messages whose IDs are 10101010101_2, we simply assign the ID filter to 10101010101_2 and the ID mask to 00000000000_2.

If we wish to filter even further – say, only those messages with an ID of 10101010101_2 ***and*** values of 0x14 or 0x15 in the first data (payload) byte – then the ID filter is set the same but we add a value (to be described below) to the Data[0] portion of the filter.

To complete Example #3, the register arrays would be programmed as follows:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	1	0	1	0	1	0	1	0
1	1	0	1	0	0	0	0	0
2	0	0	0	1	0	1	0	0
3	0	0	0	0	0	0	0	0

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
2	0	0	0	0	0	0	0	1
3	1	1	1	1	1	1	1	1

We can see from the mask register array that the RTR bit, bit 0 of Data[0], and all bits in Data[1] are to be ignored. By ignoring bit 0 of Data[0], message with 0x14 or 0x15 data values will make it through the filter. By setting the Data[1] mask to all 1s (0xFF), the Data[1] byte of the payload is ignored entirely.

Single-Filter, Extended-Frame

This option expands upon the "single-filter, standard-frame" mode. The differences are:

- All 29 bits of the **extended-frame** message ID can be tested
- The values of Data[1..0] (payload) are **not** tested

For the "Single-Filter, Extended-Frame" option, the four acceptance filter registers assume the following format:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID_28	ID_27	ID_26	ID_25	ID_24	ID_23	ID_22	ID_21
1	ID_20	ID_19	ID_18	ID_17	ID_16	ID_15	ID_14	ID_13
2	ID_12	ID_11	ID_10	ID_9	ID_8	ID_7	ID_6	ID_5
3	ID_4	ID_3	ID_2	ID_1	ID_0	RTR	n/a	n/a

And the four mask registers:

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID_28	ID_27	ID_26	ID_25	ID_24	ID_23	ID_22	ID_21
1	ID_20	ID_19	ID_18	ID_17	ID_16	ID_15	ID_14	ID_13
2	ID_12	ID_11	ID_10	ID_9	ID_8	ID_7	ID_6	ID_5
3	ID_4	ID_3	ID_2	ID_1	ID_0	RTR	n/a	n/a

Example #4

If we want our application to process ***only*** those extended-frame CAN messages whose IDs are $11001110000110101101000001111_2$ (0x19C35A0F), we simply assign the ID filter to $11001110000110101101000001111_2$ and the ID mask to $00000000000000000000000000000_2$.

If we wish to 'open' the filter a bit – say, allow all messages with an ID of 1100111000011yyyyyyyy00001111₂ (0x19C3**YY**0F), where the "don't care" bits are indicated by the 'y' placeholders – then the ID filter is set to the same value as above but we program the value $00000000000000111111110000000_2$ (0x0000FF00) into the ID mask, where there is a '1' for every 'y' placeholder in the filter ID.

To complete Example #4, the register arrays would be programmed as follows:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	1	1	0	0	1	1	1	0
1	0	0	0	1	1	0	0	0
2	0	0	0	0	0	0	0	0
3	0	1	1	1	1	0	0	0

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1
2	1	1	1	1	1	0	0	0
3	0	0	0	0	0	1	0	0

We can see from the mask register array that ID bits 15..8 and the RTR bit are to be ignored.

Dual-Filter, Standard-Frame

This option also expands upon the "single-filter, standard-frame" mode. The differences are:

- **Two** 11-bit, standard-frame message IDs can be tested
- The value of Data[0] (payload) can be tested as part of filter #1

For the "Dual-Filter, Standard-Frame" option, the four acceptance filter registers assume the following format:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID1_10	ID1_9	ID1_8	ID1_7	ID1_6	ID1_5	ID1_4	ID1_3
1	ID1_2	ID1_1	ID1_0	RTR1	Data[0].7	Data[0].6	Data[0].5	Data[0].4
2	ID2_10	ID2_9	ID2_8	ID2_7	ID2_6	ID2_5	ID2_4	ID2_3
3	ID2_2	ID2_1	ID2_0	RTR2	Data[0].3	Data[0].2	Data[0].1	Data[0].0

In this case, ID1_xx, RTR1, and **Data[0]** apply to **filter #1,** while **ID2_yy** and **RTR2** apply to filter #2.

And the four mask registers:

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID1_10	ID1_9	ID1_8	ID1_7	ID1_6	ID1_5	ID1_4	ID1_3
1	ID1_2	ID1_1	ID1_0	RTR1	Data[0].7	Data[0].6	Data[0].5	Data[0].4
2	ID2_10	ID2_9	ID2_8	ID2_7	ID2_6	ID2_5	ID2_4	ID2_3
3	ID2_2	ID2_1	ID2_0	RTR2	Data[0].3	Data[0].2	Data[0].1	Data[0].0

Example #5

By having two filters available for our application we can allow two different message types to pass thru the CAN. Let's assume that we're interested in messages that have IDs of $0010xxxxxxx_2$ – but only if Data[0] is equal to $1111zzzz_2$ – and messages that have IDs of $0011yyyyyyy_2$. All messages with the RTR bit set are to be ignored.

To complete Example #5, the register arrays would be programmed as follows:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	1	0	0	0	0	0
1	0	0	0	0	1	1	1	1
2	0	0	1	1	0	0	0	0
3	0	0	0	0	0	0	0	0
Mask[]	**bit 7**	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
0	0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0
2	0	0	0	0	1	1	1	1
3	1	1	1	0	1	1	1	1

Dual-Filter, Extended-Frame

This option also expands upon the “dual-filter, standard-frame” mode. The differences are:

- Two **16-bit,** extended-frame message IDs can be tested
- No testing of RTR bits or payload

For the “Dual-Filter, Extended-Frame” option, the four acceptance filter registers assume the following format:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID1_28	ID1_27	ID1_26	ID1_25	ID1_24	ID1_23	ID1_22	ID1_21
1	ID1_20	ID1_19	ID1_18	ID1_17	ID1_16	ID1_15	ID1_14	ID1_13
2	ID2_28	ID2_27	ID2_26	ID2_25	ID2_24	ID2_23	ID2_22	ID2_21
3	ID2_20	ID2_19	ID2_18	ID2_17	ID2_16	ID2_15	ID2_14	ID2_13

In this case, ID1_xx bits apply to filter #1, while ID2_yy bits apply to filter #2. Note that only the *upper 16 bits* – bits 28..13 – of each ID are tested; ID bits 12..0 are ignored.

And the four mask registers:

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	ID1_28	ID1_27	ID1_26	ID1_25	ID1_24	ID1_23	ID1_22	ID1_21
1	ID1_20	ID1_19	ID1_18	ID1_17	ID1_16	ID1_15	ID1_14	ID1_13
2	ID2_28	ID2_27	ID2_26	ID2_25	ID2_24	ID2_23	ID2_22	ID2_21
3	ID2_20	ID2_19	ID2_18	ID2_17	ID2_16	ID2_15	ID2_14	ID2_13

Example #6

By having two filters available for our application we can allow two different message types to pass thru the CAN. For example, one bank of IDs could be used to identify routine messages specific only to our application, while a second bank of IDs identify high-priority messages for the entire bus.

Let's assume that our routine messages have (extended) IDs of 001100001x..x_2 (0x061XXXXX) and bus-common high-priority messages have IDs of 110000000y..y_2 (0x180YYYYY).

To complete Example #6, the register arrays would be programmed as follows:

Accept[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0

Mask[]	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
3	0	1	1	1	1	1	1	1

Note that only the upper 16 bits of each ID are tested.

5.0 API mapping to Registers

The Wolverine's CAN API specifies a data structure for initializing the CAN filters, irrespective of mode. The structure is defined in wolv_can.h and called CAN_FilterInitTypeDef *[the comments and conditional compilation options have been modified/removed for brevity]:*

```
typedef struct
   {
   CAN_FILTER_MODE    CAN_FilterMode;           /*!< filter mode (single or double) */
   CAN_FRAME_FORMAT   CAN_FrameFormat;          /*!< frame format (standard or extended) */
   uint32_t           CAN_FilterID_Value1;      /*!< Specifies the ID value for receive filter #1
                                                     Bits 10..0 in std mode, 28..0 in ext mode */
   uint8_t            CAN_FilterRTR1;           /*!< RTR value for receive filter #1 (0 or 1) */
   uint8_t            CAN_FilterData[2];        /*!< Data[] values for receive filter #1 */
   uint32_t           CAN_FilterID_Value1_Mask; /*!< which ID value bits are ignored, by 1's.
                                                       Bits 10..0 in std mode,8..0 in ext mode */
   uint8_t            CAN_FilterRTR1_Mask;      /*!< RTR bit is ignored, indicated by a 0x01 */
   uint8_t            CAN_FilterDataMask[2];    /*!< which Data[] values are ignored, by 1's */
   uint32_t           CAN_FilterID_Value2;      /*!< Specifies the ID value for receive filter #2
                                                      Bits 10..0 in std mode, 28..0 in ext mode */
   uint8_t            CAN_FilterRTR2;           /*!< RTR value for receive filter #2 (0 or 1) */
   uint32_t           CAN_FilterID_Value2_Mask;  /*!< which ID value bits are ignored, by 1's.
                                                      Bits 10..0 in std mode, 28..0 in ext mode */
   uint8_t            CAN_FilterRTR2_Mask;      /*!< RTR bit is ignored, indicated by a 0x01. */
   } CAN_FilterInitTypeDef;
```

The following table indicates which fields are used for each available CAN mode:

Field	BasiCAN	PeliCAN: single/std	PeliCAN: single/ext	PeliCAN: dual/std	PeliCAN: dual/ext
CAN_FilterMode	No	Yes	Yes	Yes	Yes
CAN_FrameFormat	No	Yes	Yes	Yes	Yes
CAN_FilterID_Value1	Yes [1]	Yes [2]	Yes [3]	Yes [2]	Yes [5]
CAN_FilterRTR1	No	Yes	Yes	Yes	No
CAN_FilterData[2]	No	Yes	No	Yes	No
CAN_FilterID_Value1_Mask	Yes [1]	Yes [2]	Yes [3]	Yes [2]	Yes
CAN_FilterRTR1_Mask	No	Yes	Yes	Yes	No
CAN_FilterDataMask[2]	No	Yes	No	Yes [4]	No
CAN_FilterID_Value2	No	No	No	Yes [2]	Yes [5]
CAN_FilterRTR2	No	No	No	Yes	No
CAN_FilterID_Value2_Mask	No	No	No	Yes [2]	Yes [5]
CAN_FilterRTR2_Mask	No	No	No	Yes	No

1. Bits 10..3, bits 2..0 are shifted-off by the API
2. Bits 10..0
3. Bits 28..0
4. 4 Bits 7..0 of Data[0] only, **applies to Filter #1 only**
5. 5 Bits 28..13, bits 12..0 are shifted-off by the API

It's important to note that the API expects ID values assigned in the CAN_FilterInitTypeDef structure to be completely assigned. This means that regardless of how the ID bits are programmed into the Accept[] and Mask[] registers, the API expects all 11 bits of a standard-frame ID and all 29 bits of an extended frame ID to be assigned to the CAN_FilterID_ValueX and CAN_FilterID_ValueX_Mask fields**. The API will perform any shifting and/or truncating of ID bits.**

The following examples show how the CAN_FilterInitTypeDef structure would be initialized to accomplish the filter initialization for all six of the above examples, where:

```
CAN_FilterInitTypeDef InitStruct;
```

For any given configuration, any unused fields in the CAN_FilterInitTypeDef structure are ignored by the API.

Example #1: BasiCAN, (single-filter, standard-frame)

(Assumes MyCAN has been configured for BasiCAN…)

```
InitStruct.CAN_FilterID_Value1 = 0x56C; //                    10101101100b
InitStruct.CAN_FilterID_Value1_Mask = 0x000;
CAN_FilterInit (MyCAN, &InitStruct);
```

Example #2: BasiCAN, (single-filter, standard-frame)

(Assumes MyCAN has been configured for BasiCAN…)

```
InitStruct.CAN_FilterID_Value1 = 0x560; //                    10101100000b
InitStruct.CAN_FilterID_Value1_Mask = 0x018; // 00000011000b
CAN_FilterInit (MyCAN, &InitStruct);
```

Example #3: PeliCAN, (single-filter, standard-frame)

(Assumes MyCAN has been configured for PeliCAN…)

```
InitStruct.CAN_FilterMode          =        CAN_FILTER_MODE_SINGLE;
InitStruct.CAN_FrameFormat         =      CAN_FRAME_FORMAT_STANDARD;
InitStruct.CAN_FilterID_Value1     =         0x560; // 10101010101b
InitStruct.CAN_FilterRTR1          =                             0;
InitStruct.CAN_FilterData[0]       =                          0x14;
InitStruct.CAN_FilterData[1]       =                          0x00;
InitStruct.CAN_FilterID_Value1_Mask =                        0x000;
InitStruct.CAN_FilterRTR1_Mask = 1;                 // ignore RTR bit
InitStruct.CAN_FilterDataMask[0]   =                          0x01;
InitStruct.CAN_FilterDataMask[1]   =                          0xFF;
CAN_FilterInit                     (MyCAN,            &InitStruct);
```

Example #4: PeliCAN, (single-filter, extended-frame)

(Assumes MyCAN has been configured for PeliCAN…)

```
InitStruct.CAN_FilterMode = CAN_FILTER_MODE_SINGLE;

InitStruct.CAN_FrameFormat = CAN_FRAME_FORMAT_EXTENDED;
// 1100111000011yyyyyyyy00001111b = 0x19C3YY0F = 0x19C3000F

InitStruct.CAN_FilterID_Value1 = 0x19C3000F; InitStruct.CAN_FilterRTR1 = 0;
// 00000000000001111111100000000b = 0x0000FF00

InitStruct.CAN_FilterID_Value1_Mask = 0x0000FF00;

InitStruct.CAN_FilterRTR1_Mask = 1; // ignore RTR bit

CAN_FilterInit (MyCAN, &InitStruct);
```

Example #5: PeliCAN, (dual-filter, standard-frame)

(Assumes MyCAN has been configured for PeliCAN…)

```
InitStruct.CAN_FilterMode = CAN_FILTER_MODE_DUAL;

InitStruct.CAN_FrameFormat = CAN_FRAME_FORMAT_STANDARD; // 0010xxxxxxxb = 0x100

InitStruct.CAN_FilterID_Value1 = 0x100;

InitStruct.CAN_FilterRTR1 = 0; InitStruct.CAN_FilterData[0] = 0xF0; // 1111zzzzb

InitStruct.CAN_FilterID_Value1_Mask = 0x07F; // 00001111111b

InitStruct.CAN_FilterRTR1_Mask = 0; // filter on RTR bit

InitStruct.CAN_FilterDataMask[0] = 0x0F; // 00001111b // 0011xxxxxxxb = 0x180

InitStruct.CAN_FilterID_Value2 = 0x180;

InitStruct.CAN_FilterRTR2 = 0;

InitStruct.CAN_FilterID_Value2_Mask = 0x07F; // 00001111111b

InitStruct.CAN_FilterRTR2_Mask = 0; // filter on RTR bit

CAN_FilterInit (MyCAN, &InitStruct);
```

Example #6: PeliCAN, (dual-filter, extended-frame)

(Assumes MyCAN has been configured for PeliCAN…)

```
InitStruct.CAN_FilterMode = CAN_FILTER_MODE_DUAL;

InitStruct.CAN_FrameFormat = CAN_FRAME_FORMAT_EXTENDED;
// 001100001xxxxxxxxxxxxxxxxxxxxb = 0x061XXXXX = 0x06100000

InitStruct.CAN_FilterID_Value1 = 0x06100000; // 00000000011111111111111111111b = 0x000FFFFF

InitStruct.CAN_FilterID_Value1_Mask = 0x000FFFFF;
// 110000000yyyyyyyyyyyyyyyyyyyyyb = 0x180YYYYY = 0x18000000

InitStruct.CAN_FilterID_Value2 = 0x18000000; // 00000000011111111111111111111b = 0x000FFFFF

InitStruct.CAN_FilterID_Value2_Mask = 0x000FFFFF;

CAN_FilterInit (MyCAN, &InitStruct);
```

Revision History

Date	Revision #	Author	Change Description	Page #
8/14/19	1.0.0	SW,JA	Original app note from Scott Wright, Updated by Jose Aguas. Initial Release	
1/3/21	1.0.1	MB	Updated to new design	