Implementation of automotive CAN module requirements

Alan Devine, freescale semiconductors

At first glance all CAN modules are very similar, the only difference being the number of message buffers which are included in the implementation. Depending on the number of buffers the module is often referred to as FullCAN or BasicCAN. A FullCAN implementation normally has an array of message buffers that can be configured as Transmit (Tx) or Receive (Rx) whereas BasicCAN has a limited amount of Tx buffers and an Rx FIFO(s). However, there are several other key requirements for a CAN module that are important in the selection of the most suitable module for an application, as they can have a big impact on the efficiency of the software.

This paper discusses the main functional requirements of a CAN module for the automotive market and explores different implementations of the key requirements. Specifically, it compares implementing the requirements within a standalone module with the alternative approach of using a simpler CAN macro in conjunction with other standard MCU resources, such as system RAM, co-processor and DMA, which are not dedicated to CAN.

1 Introduction

This paper discusses the important functional requirements of message buffering, message filtering, and communications gateway (the transfer of data between different nodes on the same Electronic Control Unit (ECU)) of a CAN module that will be integrated within a microcontroller (MCU). Each of the requirements are discussed in detail and possible implementations suggested.

2 Message buffer requirements

A CAN module is known as a BasicCAN module or a FullCAN module depending on its buffer configuration. However, the message buffer (MB) configuration and total number of MB's are not the only important requirements placed on the buffers. During transmission it is important to ensure that the CAN node can transmit a stream of high priority frames without a lower priority frame, from another node, interrupting the stream. This problem is known as outer priority inversion and requires the internal processing time of the CAN module to be smaller than the minimum Inter Fame Space (IFS) to guarantee that consecutive frames from a node can be sent. Figure1 shows the outer priority problem. In this example the lower priority frame1 (Node2) is inserted between the higher priority frames 1 and 2 (Node1) on the CAN bus, as the internal processing time of node1 is greater than the IFS.

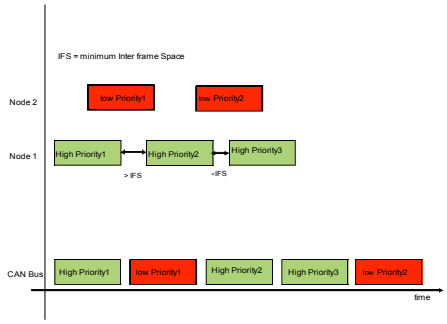


Figure 1 - Outer priority inversion

In order to avoid this problem the MB's needs to be loaded immediately after transmission and typically requires more than a single transmit MB to decouple the reloading procedure from the current transmission. The decoupling is also required to increase the amount of time that the CPU has to reload the MB's. Without this decoupling it would be very difficult to guarantee that the software can reload the MB's fast enough. Normally a

minimum of 3 transmit MB's are required to cover all circumstances, as problems can arise if only 2 MB's are used. e.g. If the sending of a message is finished and the second buffer is still being reloaded.

The module must also ensure that the highest priority scheduled frame is transmitted first onto the bus. Each Tx MB needs to have an internal priority mechanism to allow the highest priority frame the opportunity to fight for bus arbitration. This is particularly important for the transmission of Transport Protocol (TP) messages which can have the same ID, but need to be transmitted in chronological order. The module also must not block high priority messages from transmission by lower priority scheduled messages already loaded in the Tx MB's. This problem is known as inner priority inversion and is shown in figure 2. This can occur when the other nodes on the network are transmitting higher priority frames (lower ID) than the locally scheduled frames. Subsequently, if all Tx MB's of the local node are full and blocked from transmitting due to the bus traffic, the transmission of a new higher priority frame (local node) is then delayed until a scheduled frame is successfully transmitted. In order to avoid this from happening hardware cancellation needs to be supported on the Tx MB's to allow the new higher priority frame to be loaded into a buffer and allowed to fight for bus arbitration.

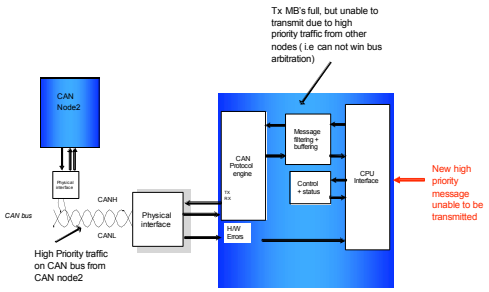


Figure 2: Inner priority inversion

The receive path is just as critical as the transmit path. In the ideal world the number of receive MB's should be equal to the number of unique messages to be received and each Rx MB's should have an associated queue to allow the CPU more time to service the MB before data is lost. However, as all applications are not identical it is impossible to fix the number of MB's that would match all circumstances. Typically, FullCAN implementations with 16, 32 and 64MB's variants are available, as the cost of modules with more dedicated MB's becomes prohibitive. If more frames than MB's exist, then multiple frames need to be received in some of the MB's, which is possible using hardware filtering (see following section). Several possible solutions exist to extend the number of MB's per node.

One possible solution is to use the MCU's system RAM for the MB's, which allows the number of MB's to be configured at the expense of the RAM. This is a very attractive approach due to the flexibility, but as the number of CAN modules on a single MCU increases, 6 at the last count, this becomes more difficult to handle and can lead to system performance issues. The MCU architecture also has to be able to allow the CAN module(s) to be a bus master so that they can directly access the system RAM. This will inevitably require a new CAN module design.

An alternative solution is to develop a CAN module that can share a common pool of local RAM (MB's). Thus, the number of MB's per module can be optimized to the application requirements. For example a node on the bus may require 48MB's and another only 16MB's. This could be satisfied with a total pool of 64 MB's. If a shared pool of memory is not used, It would require a 16MB's FullCAN and 64MB's FullCAN module, which results in 80MB's. This example assumes only 16, 32 and 64MB's FullCAN's are available. It is also typical that multiple CAN modules on a MCU have the same amount of MB's. In the worst case two 64MB's CAN modules would be used, resulting in a total

of 128MB's when only 64 are required. Thus, with a pool of memory approach it would be possible to support the requirements of both nodes with fewer buffers which would result in a smaller CAN module.

A further solution is to offer a BasicCAN implementation and use an on chip DMA or co-processor to build the MB's in system RAM. This is a very flexible option, however to be truly effective the DMA should be coupled with a CAN module with a perfect receive filter (see next section) and the CAN module needs to be able to operate with the DMA. If a co-processor is used the hardware filter can be simplified, as the co-processor can efficiently handle software filtering and it should be possible to reuse an existing CAN module.

A further requirement on the receive buffers is the ability to 'queue' the received frames to increase the amount of time the CPU has to handle the reception without the loss of data. If messages are queued it is important that the chronological order is kept, thus a hardware FIFO is normally the best solution. However, other approaches which are more flexible than a hardware FIFO could also be implemented. For example several MB's could be joined together to make a queue. This solution offer's the ability to customize the number of Rx queues and depth of queues at the expense of individual MB's. The second approach could be a hardware linked list where the queue is dynamically allocated from a pool of Rx buffers. Both of these approaches are attractive as they offer more flexibility than the FIFO solution. The main drawback would be additional software overhead compared to a fixed FIFO and the module complexity.

3 Message filtering requirements

One of the most important requirements for any CAN module targeted for automotive applications is its ability to reject frames that are not intended for the particular node. This is known as message filtering and is important as it can vastly reduce the number of interrupts that a CPU has to handle. Essentially, the message filter block compares an incoming frame's arbitration field against a preprogrammed filter value that is normally configured at initialization. If the incoming message's arbitration field matches the filter value, the entire frame is stored in the CAN modules hardware Rx MB and an interrupt request is sent to the CPU (if enabled). If a match does not occur the frame is not copied into a MB and an interrupt request is not generated. The Bosch CAN specification (CAN2.B) optionally includes mask bits that allow any ID bit to be set as don't care in order to accept groups of identifies to be received. Most controllers have this basic level of message filtering, however, due to the large number of ID's used within modern CAN networks, this basic filtering is not always capable of accepting only the frames intended for the specific node. This leads to the leakage of unwanted frames through the filter which increases the interrupt and CPU loading, as the CPU needs to perform secondary software filtering on all messages. Figure 3 demonstrates this filtering concept.

	Base ID ID28……ID18	IDE	Extended ID ID17……ID0	Reception Status
MB ID	00001111111	1	00000000000001111	
RX Mask	11111111111	1	11111111111111100	
RX Frame 1	00001111111	1	00000000000001100	Accepted
RX Frame 2	00001111111	1	00000000000001101	Accepted
RX Frame 3	00001111111	1	00000000000001110	Accepted
RX Frame4	00001111111	1	00000000000001111	Accepted
RX Frame5	00001111111	1	00000000000001011	Rejected

Figure 3: Bitwise masking example

It can be seen for the example that frames with ID's in the range 0x01FC000F to 0x01FC000C are accepted as ID bits 0 and 1 are set to don't care (00). With this approach it is relatively simple to receive groups of frames. However, this filter concept is not 100% effective (reject all

unwanted frames). For example it is not possible to only accept the contiguous range ID3 to ID13. (See figure 4 for a further example)

In automotive networks several types of messages are present and the different message types are often grouped into separate ranges. For example it is common to find application messages, diagnostic messages, network management messages and transport protocol messages. The latter 2 types can usually be filtered with the bitwise masks discussed above. The application messages are more difficult to filter, as they can be grouped into a mixture of bitwise ranges, contiguous ranges and a collection of individual ID's. Figure 4 shows a sample of frame ID's used within a real application.

Message ID	Filter type	Filter Example
0x280	Bitwise Mask	ID - 0x280
0x288		MSK- 0x3F7
0x440	Bitwise Mask	ID - 0x440
0x540		MSK- 0x6FF
0x5E0	Look Up Table(LUT)	0x5E0
0x271		0x271
0x3E1		0x3E1
0x389		0x389
0x300 - 0x30F	Bitwise Mask	ID - 0x300
		MSK - 0x3F0
0x693	Contiguous Range	START - 0x693
0x694		
0x695		
0x696		
0x697		END - 0x697

Figure 4: Filtering example

It can be seen from the example that the bitwise mask approach specified within the CAN specification is not flexible enough to accept only the frames for the specific node. A mixture of bit masks, contiguous range and a LUT are necessary to create a 100% effective filter. Note: If the LUT was big enough there would be no need for the other filter types, however this would lead to excessively large LUT. Another solution is to use a FullCAN implementation where the number of Rx MB's is greater than the number of frames to be received. In this case each MB accepts a frame with a unique ID and since there are more MB's than frames to be received, only the intended frames are accepted. However, there is normally a limitation to the number of MB's that are implemented due to the additional cost. In the case that there are more frames than MB's it is still possible that unwanted frames are received. Although, this possibility can be reduced by including bit filter masks on each MB, which enable a particular range to be received. The main problem is that there is not a standard definition of CAN frame ID's in the automotive industry and that it would be excessively expensive to build a hardware filter block to cover every situation.

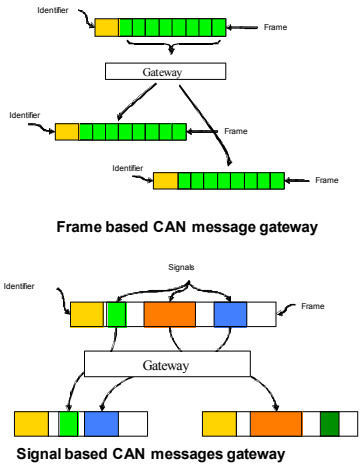
There exist several different approaches to solve the filtering problem. The first is to keep adding extra hardware filters, which can be configured for bit masks, contiguous ranges and the inclusion of a LUT for the individual identifiers. This has the main advantage of simplicity to the user and should be more cost effective than a single large LUT. The main disadvantages are inflexibility and cost. The silicon manufacturer still has to make assumptions about the maximum number of filters to include and invariably there will be applications that require more flexibility than offered.

The second option is a subtle variation on the first. Essentially, the same filter blocks are used, but they are shared across multiple CAN channels which in theory should lead to a smaller amount of blocks and reduced size. The idea is that different channels require different amounts of filtering for different applications, thus it is possible to have very flexible filtering on one channel at the expense of another channel. This is attractive as the total number of filter blocks required, to achieve the same level of filtering, should be less than the previous solution. However, to be effective the CAN module needs to support multiple channels, which adds to the design complexity that has implications on test and assumptions still need to be made for the amount of filter blocks to include.

The third option is a software approach that uses a co-processor that has been specifically developed to quickly handle interrupts. This is the only approach that offers a 100% effective filter for all applications. The co-processor executes the software filter algorithm and only interrupts the main core when a successful match has been detected. In addition the co-processor can perform more than the filtering algorithm, for example servicing of the hardware which is wasted effort for all unwanted frames. The main disadvantage with this approach is the added complexity to the hardware, the software which needs to be taken into account and the small percentage of bus bandwidth that will be used when the co-processor is executing.

4 Communication gateway requirements

Another function that is becoming more important, particularly in the automotive area, is the CAN gateway. In a typical car there are between 3-6 CAN networks connected by a gateway node. The sharing of data between networks is referred to as a gateway. Gateways are normally a mixture of frame gateways (the entire frame is copied to another network without changing content of the frame) and signal gateways where selected signals (data within the frame) are copied into one or more frames. Figure 5 shows an example of a frame and signal based CAN gateway.



- All signals for a gateway frame do not always arrive at the same time
- The same signals can be copied to several gateway frames.

Figure 5 – Frame and signal based gateway

If several signals need to be copied from a source message to a destination frame or frames this can consume a high percentage of the CPU bandwidth, as many masking and shifting operations are required. Adding specific gateway functionality to a CAN module can help to off load the CPU when performing these gateway tasks. In addition there have been cases where OEM's have mandated that parts of the gateway function needs to be performed without the main CPU's intervention. An example of this is a simple mirror function that autonomously copies frames from any bus to the diagnostic bus.

Different methods of adding gateway functionality to a CAN module are possible. The first is to add a hardware wrapper that can be considered to logically sit above the CAN modules. In this method the wrapper accepts frames from each CAN module and performs the frame and signal gateway autonomously. This method has the advantage that it can be made to work without having to fully redesign the CAN module and as the gateway functions are performed in hardware it does not consume any CPU bandwidth. The main drawback is that the wrapper would be fairly complex to handle all possible combinations of signal gateways.

The second method is to introduce gateway functionality into the module itself, which only makes sense if the CAN module can support multiple CAN channels. The benefit of this approach is similar to the first in that a hardware approach does not consume CPU bandwidth. The main disadvantages are that a new CAN module needs to be developed, that CAN module would be complex and the gateway functionality is restricted only to CAN.

The third method is to add a small co-processor to the MCU that can be programmed to perform the gateway

functionality. As this solution is fully user programmable it is the only method that can ensure all gateways can be handled independent from the main core. It has the advantage that an existing CAN module can be used without redesign and the co-processor could be used to gateway more than CAN frames. Several other networking protocols are becoming prevalent within the automotive industry and signals from each network need to be combined. It will become typical for automotive gateways to combine CAN, LIN and FlexRay signals. For example the current AutoSAR[1] standard includes a CAN/LIN/FlexRay gateway specification. The main disadvantage of the co-processor is that it shall consume a finite percentage of the memory bandwidth and there is additional software overhead.

5 Conclusions

Automotive customers place specific requirements on the CAN module and even within the same customer the requirements can be different, depending on the application. (e.g. Body, Gateway, Powertrain, etc). This makes it very difficult for a single CAN module to be the best fit for every application. From a silicon design point of view it is very important to build as much flexibility into the module as possible and at the same time keep the die size small. Driver software also needs to be considered, particularly with the introduction of AutoSAR, as there no point building in many hardware features that will never used.

Considering the requirements discussed in this paper the Tx buffers need to be such that inner and outer priority inversion is avoided. This requires hardware support for local priority, hardware cancellation and buffer decoupling. On the receive side a hardware FIFO should be included to increase the time the CPU has to receive frames and to guarantee chronological order is maintained. It is also desirable to have a module that can be configured in a BasicCAN and/or FullCAN mode. In terms of filtering it is very difficult to build a hardware filter that can reject all unwanted message frames for every application. Adding a co-processor solves this problem as a perfect filter can be constructed in software which will only interrupt the main core when a valid frame is received. The co-processor can also be used to add flexibility to the buffer management as extra Rx and Tx buffers can easily be emulated in system RAM. Furthermore, it is very attractive solution for the gateway as all routing combinations can be built in software and other protocols can be combined. The solution of using a co-processor and a small CAN module, with required hardware support to avoid priority inversion and reception handling, offers a very flexible solution that is suitable for many different applications.

[1] AUTOSAR (AUTomotive Open System Architecture) is a development partnership. The objective of the partnership is the establishment of an open standard for automotive E/E architecture