AN1776

Stereo Audio Transmission Over The CAN Bus Using The Motorola MC68376 With TouCAN Module

By Allan Dobbin
Transportation Systems Group
East Kilbride, Scotland

Rev 1.0, 10th July 1998

1 Introduction

The main purpose of this application note is to provide the reader with a working knowledge of the Motorola TouCAN module. A non-typical CAN application of stereo-audio transmission is used as an example. The Motorola MC68376 microcontroller with TouCAN is used and its QADC and QSPI modules are demonstrated also. The source code for the TouCAN module and all audio transfer is provided in C.

In this example, an MC68376 MCU samples the stereo audio data using the QADC. The resultant sampled data is transmitted onto the CAN bus by the TouCAN module and is received by a second 68376 MCU also connected to the CAN bus. The receiving MCU uses a small internal RAM buffer for temporary storage before outputting the audio data on the QSPI to an external DAC for reproduction of the analogue signal. The MCUs use the queuing mechanism of the QADC and QSPI to minimize CPU overhead.

2 Contents

3 Summary of CAN

The Controller Area Network (CAN) was originally developed by BOSCH Gmbh as a serial communications protocol to pass information between controllers on an automotive network and thus reduce the growing complexity of the wiring harness on modern car design. The protocol had to include prioritization of messages, flexible configuration, multicast reception, multiple bus masters, error detection, fault confinement and automatic retransmission.

Over recent years there has been a steady growth in the number of applications using the CAN interface. It is already widely used in the Automotive industry in Europe and increasingly in the USA. In automotive electronics, engine control units, anti-lock braking and sensors may be connected using a high-speed CAN bus with bit-rates up to 1 Mbit/s whereas electric windows and vehicle lighting may be connected to a low speed bus with data transmission rates of between 10 to 100 kbit/s. Other applications include industrial and nautical equipment, medical apparatus, elevator controls and even entire manufacturing plants may interface intelligent control systems communicating in real time using CAN networks.

The original CAN specification provided 11 identifier (ID) bits. The updated CAN specification provides for either 11 ID bits or for a larger identifier range using 29 bits. The 11-bit ID format is referred to as the *standard* format and is governed by the CAN standard 1.2/2.0A, whilst the 29-bit ID is referred to as the *extended* format. CAN standard 2.0B caters for both 11 and 29-bit ID's. The majority of current applications use the standard 11-bit identifier as it has greater throughput i.e. the smaller ID field is less of an overhead than a 29-bit field. See reference #1 for the full CAN specification.

3.1 The Physical Layer

The physical layer covers the transfer of the data between the different nodes on the CAN network and all electrical properties. The actual physical interface is not described within the CAN specification but usually consists of a two-wire differential bus with each signal designated CANL and CANH. The 'off' state of the bus is called *recessive* and the 'on' state is *dominant*. The physical interface is designed so that more than one node may drive the bus at any one instant without damage. If both dominant and recessive bits are transmitted onto the bus together, the resulting bus state will be dominant. Using a wired-AND implementation as an example, logic "0" would be dominant and logic "1" would be recessive and the bus state would be 0 if any input was 0.

The actual voltage levels that appear on the CAN bus are not defined in the Bosch specification and several different transmission mediums may be used. Although twisted pair is common, others may be used including optical and single wire implementations. Most controller ICs implementing the CAN protocol do not include the physical layer drivers, but have CMOS transmit and receive pins instead. A common interface IC is the Philips 82C250 transceiver which implements the physical layer as defined by ISO 11898 (CAN standard for high speed communications). This IC is used in the hardware described later in this application.

The 82C250 outputs nominal voltage levels of 3.5V on CANH and 1.5V on CANL for the dominant state i.e. 2V differential. The pins do not drive during the recessive state and are tied together using termination resistors to produce a zero differential. The transceiver interprets an input differential greater than 1V as dominant.

The TouCAN module has one receive input, CANRx0 and two transmit outputs, CANTx0 and CANTx1. The transmit outputs may be configured as full CMOS with positive or negative polarity or as open drain. The receive input may be configured as 0 or 1 dominant. On the 68376 device, only CANTx0 and CANRx0 are bonded out to pins. The TouCAN module is described in detail in the next section.

The maximum speed of a CAN bus, according to the standard, is 1 Mbit/s. Since the arbitration scheme requires that the data propagates to the most remote node and back, the maximum cable length at this transmission rate is limited (by the speed of propagation) to 40 meters. The cable length may be increased at lower transmission rates i.e. up to 500 meters at 125 kbit/s. This transmission speed limitation is due to the propagation delay and is described in Section 3.5 Bit Timing.

3.2 Message Transfer

Four different frame types are possible in CAN.

A **remote frame** is transmitted from a node acting as a receiver and is a request for data. This remote frame contains no data, but does contain the ID of the type of data it is requesting.

An **error frame** is transmitted by any node on the bus which detects a bus error. This error frame consists mainly of dominant bits which override the current message and break the CAN bit stuffing rule (described in Section 3.3). This forces the transmitter to abort transmission of the current message and ensures that all nodes on the bus are aware of the error.

An **overload frame** is used when a node requires a delay between reception of successive data frames.

A **data frame** is a regular message frame that carries data from the transmitter to the receivers. Under normal operating conditions, the data frame is the predominant, or only, frame on the bus and is described in more detail. A data frame consists of several fields as shown below.

NOTE: *Data and remote frames are issued by the CAN controller after instruction by the CPU whereas error and overload frames are usually issued automatically by the CAN controller independently of the CPU.*

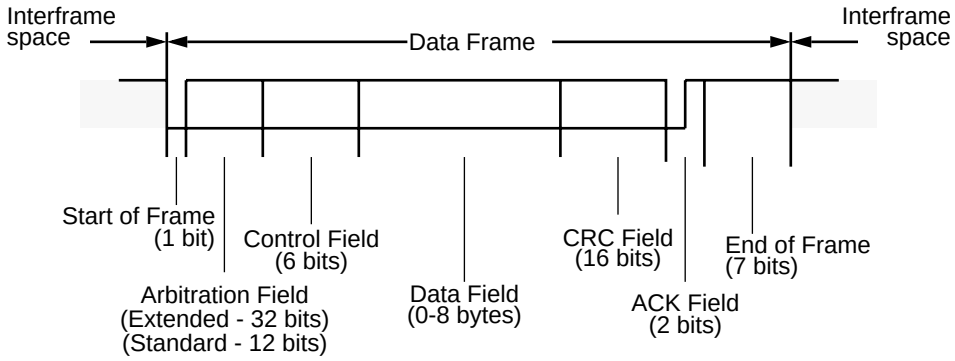

Figure 1 CAN Data Frame

start of frame — a single dominant start bit indicates the start of a message from one or more nodes.

arbitration field — this contains the message ID plus the RTR bit. An 11-bit ID is used for standard format and 29 bits for extended format as shown in Figure 2. The extended format additionally contains the SRR and IDE bits. The ID is usually used to assign the message type i.e. air temperature, engine speed, audio left, etc. and not the destination address, however this is specified by the system designer. The ID is also used to assign priority to messages – priority is discussed in more detail in the section on arbitration. The RTR bit (Remote Transmission Request) specifies whether the frame is a data frame or a remote frame. The IDE bit (Identifier Extension indicates 11-bit or 29-bit identifier) and SRR (Substitute Remote Request) are transmitted recessive for an extended format frame.

control field — this field contains a 4-bit data length code (DLC) plus two reserved bits. The DLC specifies how many bytes of data are contained within the data frame and 0 to 8 bytes are allowed. Codes for 9 to 15 are not permitted.

data field — the data consists of between 0 to 8 bytes and can vary in length i.e. air temperature may consist of two data bytes, whereas audio

may consist of eight. A zero byte message does have its uses and could be used as a wake-up command, a synchronization command or a request for data (in the case of a remote frame where RTR would be recessive).

CRC field — this consists of a 15-bit cyclic redundancy sequence which allows all nodes to perform a frame security check for detecting bit errors, plus a single recessive bit used as a CRC delimiter. The CRC sequence is derived from all preceding fields.

ack field — this consists of two bits – the ACK slot and the ACK delimiter. The transmitting node sends two recessive bits but any other active node on the network which receives the frame correctly will force the first bit to dominant. This acts as an acknowledgment to the transmitter that the message has been successfully transmitted on the bus. This does not necessarily mean that the node(s) which the message was intended for has received it, but only that at least one node has received it – all active nodes will acknowledge the frame whether or not they have been programmed to filter that particular ID. If the transmitting node does not detect a dominant acknowledge bit, it will repeat transmission of the frame.

End of frame — the end of message is indicated by the “end of frame” field which consists of seven recessive bits.

Interframe space — this normally consists of an intermission period of 3-bit times plus the bus idle period. Nodes cannot transmit data frames or remote frames during the intermission period and the bus will be recessive during this 3-bit period. The bus idle period is of arbitrary length and all nodes will recognize the bus as being free during this recessive state.

The arbitration fields for standard and extended formats is shown in Figure 2.

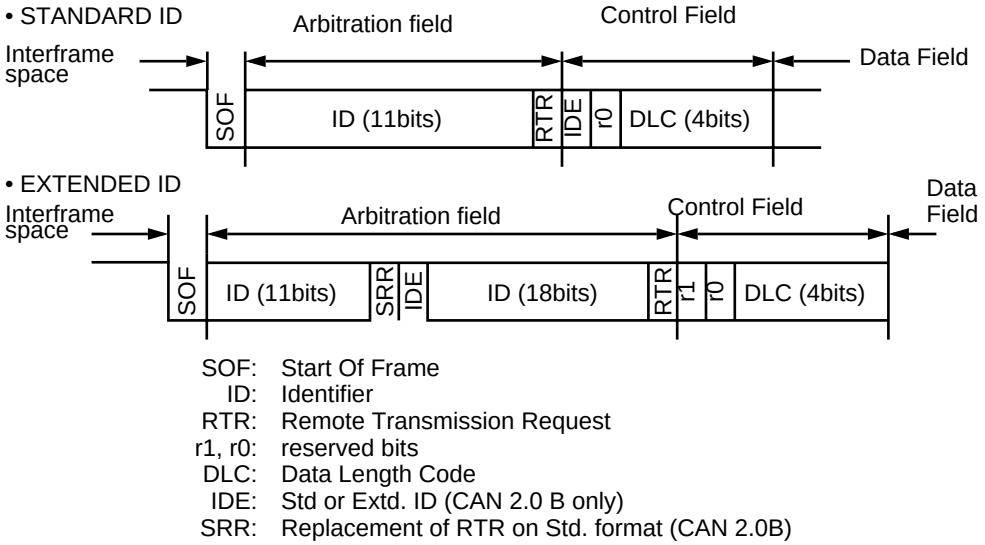

Figure 2 Standard vs Extended Arbitration Fields

3.3 Monitoring and Arbitration

There are certain rules that all nodes must adhere to in keeping with the CAN protocol standard. These include bit stuffing, cyclic redundancy check, frame checks and the acknowledgment process. Bit stuffing ensures that sufficient edges are generated for synchronization since NRZ (non-return-to-zero) coding is used – after five consecutive equal bits, an additional complementary bit is added, or stuffed, by the transmitter. To enforce these rules all nodes must monitor the bus. This means that all active nodes on the network, receivers and transmitters alike, will monitor every bit of each message for proper conformance to the CAN protocol. As soon as violation is detected by any node then those nodes will transmit an error frame which consists of six consecutive dominant bits (except in the case of error passive nodes which transmit a recessive error frame – see Section 3.4 Errors and Fault Confinement). Since this violates the law of bit stuffing, it acts as a message to all other nodes that the current message is corrupt and transmission should be aborted.

This monitoring of bits by the transmitter serves another purpose i.e. arbitration. Since several nodes may commence transmission at the same time, each message's unique ID within the arbitration field is used to determine who wins control of the bus. The process is as follows: two or more nodes transmit a dominant start bit at the same time and both successfully monitor the correct state on the bus. They then commence transmission of the 12 (or 32) arbitration bits until at some point one node transmits a dominant and the other transmits a recessive. Due to the nature of the physical interface, the dominant bit prevails and so the

unsuccessful node detects this and simply backs off its own transmission and waits for the current message to end before trying for bus arbitration again, but participates as a receiver on the current transmission. The successful node continues transmission and is unaware that there was ever any conflict, or arbitration, for the bus.

The arbitration process allows assignment of priority by giving highest priority messages a low numerical ID (in a 0 dominant system). Generally, high priority is assigned to rapidly changing data. All CAN nodes receiving a message must check the ID to determine whether or not there is any useful data in the message for its use. On highly integrated CAN controllers (such as the TouCAN), the hardware will provide ID filtering where a mask permits only certain ID combinations to filter through to the internal receive buffers. Less integrated controllers will not provide hardware filtering thus the application software will have to burden the CPU with the additional task of testing the message ID of every message received on the bus.

3.4 Errors and Fault Confinement

Fault confinement on the bus is possible as CAN nodes are able to distinguish between short disturbances and permanent failures. This is accomplished by the use of two 8-bit error counters within each node – a transmit error counter and a receive error counter. In summary, the node will increase its receive or transmit counter when it detects any of the five possible error types (bit error, stuff error, CRC error, form error or acknowledgment error) and will decrement when it completes a successful reception or transmission. This means that any temporary disturbances result in small counts that recover back to zero, whereas permanent failures result in large counts.

The Rx error counter increases by 1 if an error is detected during reception. However if the first bit after transmission of an error flag is dominant, which suggests that another node did not detect this same error, the Rx error counter is increased by 8. The Tx error counter is always increased by 8 if an error is detected while the node is transmitting. The Rx error counter is decreased by 1 after a successful reception and the Tx error counter is decreased by 1 after a successful transmission. This method of incrementing the counters ensures that if the fault is local to a node then only its own error counters will increment rapidly.

A node will take on one of three states depending on the value within its error counters as follows:

error active — this is the regular operational state of the node and occurs when both counts are less than 128. In this state the node can participate in usual communication. If it detects any errors during

communication, it indicates this by transmitting an ERROR ACTIVE FLAG which consists of 6 dominant bits and therefor blocks the current transmission.

error passive — this state occurs when either of the counters increment past 127 and indicates that there is an abnormal level of errors at this node. The node still participates in transmission and reception, but is forced to wait slightly longer after a message transmission before it can initiate a new message transfer of its own. This extra delay for the error passive node is known as *suspend transmission* and is accomplished by having the node send an additional 8 recessive bits at the end of its frame. This means that an error passive node loses arbitration to any error active node regardless of the priority of their IDs. When an error passive node detects an error during communication it indicates this by transmitting an ERROR PASSIVE FLAG. This consists of 6 recessive bits which will not disrupt the current transmission (assuming another node is the transmitter) if the error turns out to be local to the error passive node.

bus off — this state occurs when the transmit error count reaches 256. This indicates that the node has experienced consistent errors whilst transmitting. In this state, the node switches off its bus drivers and no longer influences the bus. The node will eventually be re-enabled for transmission and become error-active after it has detected 128 occurrences of 11 consecutive recessive bits on the bus which indicate periods of bus inactivity.

3.5 Bit Timing

Nodes connected to the CAN bus use a high frequency clock and a prescaler of at least 5 bits which allows division of this clock by a range of at least 1 to 32. The period of this resulting lower frequency clock is a *time quantum* and this is the basic unit of measurement for the CAN bit timing. The transmission period of the node will be a multiple of these time quanta.

Referring to Figure 3, each bit on the CAN bus is composed of four time segments as follows:

synchronisation segment — this has a fixed size of one time quanta and the bit edge is expected to lie within this segment. Each node synchronizes to the transmitting node by ensuring that the first edge of a message lies within this segment. Further resynchronizations are performed on subsequent edges within the message.

propagation segment — this is programmable between 1 and 8 time quanta and compensates for delays on the CAN network. Its value must be at least twice the maximum time the signal takes to propagate between any two nodes on the system, i.e.

$$prop > 2 \bullet (\text{output driver delay} + \text{bus line delay} + \text{input comparator delay})$$

The output driver and input comparator delays are dependent on the bus interface hardware and the bus line delay is the total bus length multiplied by the speed of propagation (approx. 2 x 10^8 m/s but depends on cable used).

phase segment 1 and 2 – pseg1 is programmable between 1 and 8 time quanta and pseg2 will be of at least equal length, usually down to a minimum of 2. The minimum value of pseg2 is required to allow the node time to process the bit value since it is sampled at the end of pseg1. This minimum process time is called the information processing time or IP. The CAN node will automatically lengthen pseg1 or shorten pseg2 upon detecting edges within the message to allow resynchronization with the transmitting node. The amount of time quanta these segments may be altered by must not be greater than the *resynchronization jump width*, RJW, which is limited in size to that of pseg1 up to a maximum of 4 time quanta.

The automatic adjusting of pseg1 and 2 by the CAN controller is required to allow each node to resynchronise with the current transmitter during each recessive to dominant transmission. If the transmitter has an internal clock that is slightly slower than the receiver then the transition will be ‘late’ and will occur after the sync. seg. within either the prop or pseg1 segments. In this case, pseg1 is lengthened to compensate. Conversely, if the transmitter is fast then the transition will be received ‘early’ i.e. before the sync. seg, within pseg2 of the previous bit time. In this case, pseg2 is shortened to compensate.

The values assigned to segments are dependent on the CAN bit rate required plus the oscillator tolerance of each node. Generally, the propagation segment is set to the minimum amount of time quanta that allow for twice the worst case signal propagation delay. If maximum bit rates are required, then set pseg1 and 2 small with respect to prop. seg. If maximum resynchronization is required to allow the use of low tolerance clock oscillators then set pseg1 and 2 large in relation to prop. seg. up to a value of 4. Usually RJW is set to maximum value i.e. equal to pseg1 (up to a maximum of 4).

Refer to Section 4.9.2 for an example of TouCAN bit timing set-up.

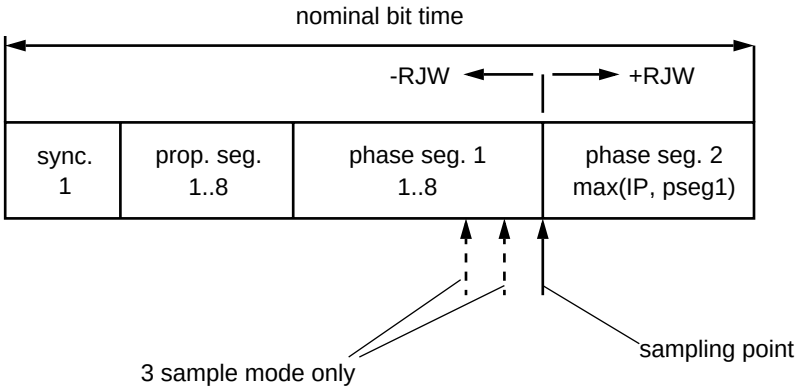

Figure 3 Can Bit Timing

3.6 Motorola CAN Modules

Motorola currently offer three different CAN modules to support CAN on each of their 8, 16 and 32-bit microcontroller families i.e. MCAN, MSCAN and TouCAN.

MCAN provides the lowest performance / highest CPU overhead of the three and is available on the HC05 family. One Tx and two Rx buffers are provided.

MSCAN has a higher level of integration and provides three Tx buffers and two Rx buffers. This module is available as two slightly different versions for the HC08 and HC12 families. The MSCAN12 version has 2 x 32-bit filtering which is double that of the MSCAN08.

TouCAN offers the highest performance and is currently available on the HC16, 683xx and PowerPC MPC500 families. It provides 16 message buffers each configurable as Rx or Tx and 3 x 32-bit filter masks. The TouCAN operation is covered in more detail in Section 4.9.

4 MC68376 Overview

The Motorola MC68376 is a member of the highly integrated 683xx modular family of microcontrollers where modular building blocks are shared throughout the family and connected internally via the inter-module bus (IMB). The 68376 comes in 160 QFP package and consists of several modules as described in the following sections. All modules are described briefly with the emphasis on the TouCAN module. For a full description refer to reference #2.

4.1 CPU32

32-bit architecture based on the 68020 processor. May operate on 8-bit, 16-bit and 32-bit operands, however only a 16-bit data bus exists outside the CPU. Contains eight 32-bit data registers, eight 32-bit address registers, 32-bit stack and program counters plus an 8-bit condition code register. Features a built-in background debugging mode (BDM) which was used in the debugging and testing of this application. The CPU32 provides seven levels of interrupt priority and 256 exception vectors.

4.2 QADC

10-bit analogue to digital converter with 16 direct input channels expandable to 44 in multiplexed mode and conversion time around 8 micro-seconds. The distinctive feature on this ADC is its queuing mechanism. The CPU does not have to intervene after every analogue conversion, instead a queue of up to 40 conversion commands may be utilized. Each conversion command has a corresponding result word register which stores the resulting 10-bit digital conversion. Our system makes extensive use of this queuing mechanism and interrupts so that there is minimal overhead on the CPU during audio sampling.

4.3 QSM

The Queued Serial Module consists of two independent ports – an SCI (Serial Communications Interface) and QSPI (Queued Serial Peripheral Interface). The SCI is a standard asynchronous serial interface, or a UART and is not used in our system. The QSPI is a synchronous interface similar to the Motorola SPI used on 8-bit micros, but with two important additions. The first of these is a sixteen word queue offering similar advantages as the queuing mechanism on the QADC. A table of up to sixteen serial transfer commands may be used which act on a corresponding table of sixteen receive and transmit data registers. The use of this queue allows reduction on the CPU overhead since up to sixteen transfers without CPU intervention are possible. The second addition is a port of four PCS (Programmable Chip Selects). Each of the sixteen queued command words activates the required PCS for one of the serial transfers. Our audio system makes use of both the queue and the PCS as will be described in the audio section.

4.4 7.5K SRAM

There are two RAM blocks – a 4K byte block of SRAM for general purpose use plus a 3.5K byte block for either general use or TPU microcode emulation. The internal SRAM is sufficient for our application, so no additional external memory is required.

4.5 8K ROM

8K bytes of masked ROM available for high volume users. In most applications this is used for initialization and start up routines.

4.6 TPU

The TPU is essentially an internal co-processor with sixteen dedicated input/output channels for control of timer based functions. This module is very useful in engine management and motor control applications but can be put to many other general purpose uses i.e. additional SCI (UART) ports or PWM channels. The TPU is not required in our application.

4.7 CTM

The configurable timer module with two 16-bit modulus counters, 16-bit free running counter, four double action submodules and four pulse width modulation submodules provides additional timer / counter functionality. The CTM is not required in our application.

4.8 SIM

The System Integration Module includes the external bus interface, 12 chip selects, system protection block with software watchdog, periodic interrupt timer, bus monitor and the PLL generated system clock.

4.9 TouCAN

The TouCAN module is a communication controller that implements the CAN protocol up to maximum possible CAN transfer rates of 1Mbit/s. Both standard (11-bit identifier) and extended (29-bit identifier) message formats are supported as specified in CAN protocol specification 2.0B. The TouCAN module includes the following features:

- 16 message buffers for receiving or transmitting data frames.
- Programmable bit rate up to 1Mbit/s.
- 2 serial message buffers for double buffering of both received and transmitted data.
- 16-bit free-running timer provides time-stamp.
- Supports CAN 2.0B – both standard and extended ID formats.
- CPU overhead reduced by implementing local ID fields within each message buffer plus ID bit masking using one of three 32-bit mask registers.
- Automatic reply mechanism available for remote request frames.
- 19 maskable interrupt sources.
- Low power sleep mode with wake-up mechanism.

4.9.1 Message Buffers

The TouCAN provides sixteen message buffers (MBs), each of which can be assigned either as a transmit or receive buffer and each may optionally generate an interrupt after successful completion of data transfer. Each buffer contains eight 16-bit registers which take two slightly different formats depending on whether standard or extended format is selected as shown.

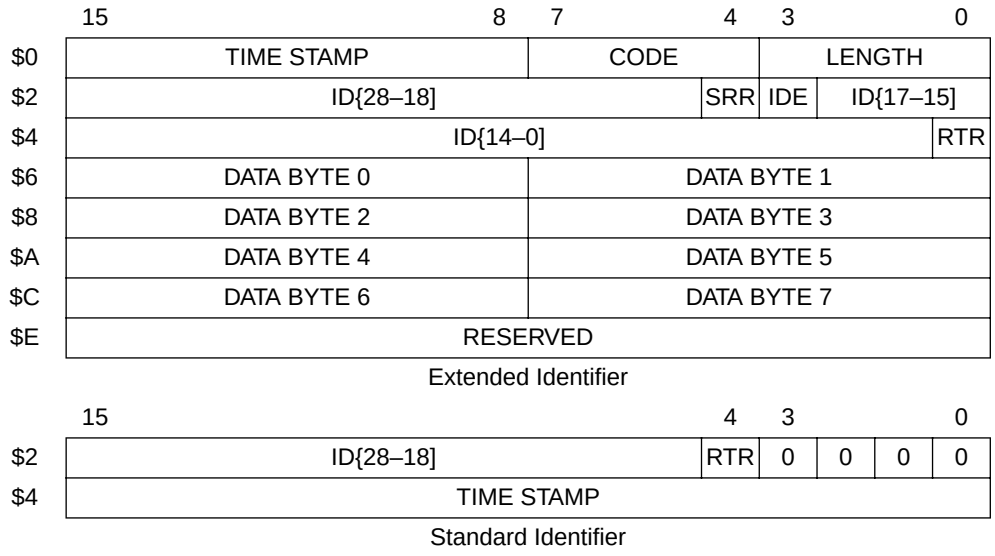

Figure 4 TouCAN Message Buffer Structures

The **code** field is initially written by the CPU to define the operation of the MB i.e. inactive, receiver, transmitter, remote request, etc. and is updated by the TouCAN after bus activity to indicate whether a successful operation has occurred. The **time stamp** is copied from the free-running timer to the MB by the TouCAN to indicate when reception or transmission occurred. On MBs configured for transmission, the CPU writes the **length** and **data** fields to be transmitted on the bus within a data frame, whereas on receive MBs, the TouCAN writes these two fields as they were received from the bus. The **ID** field is written by the CPU including the **IDE** bit to select either standard or extended format for both transmission and reception. On a Tx MB, the ID field is transmitted as written, whereas on a Rx MB, the ID field is combined with one of the receive mask registers and the resulting filter is used to check for a comparison on any received frames. If a match is made on a received frame, its ID will then be copied into the MB's ID field, regardless of the contents of the mask register.

The code field is not initialized after power-up or reset and must manually be set to an inactive receiver (0000) or inactive transmitter

(1000) by the CPU. Once all TouCAN registers have been initialized and the message buffers have their data and ID fields configured, the code may be set to active receiver (0100) or active transmitter (1100 and RTR = 0) or alternatively one of the remote modes may be selected for transmission. The remote modes are remote transmission request (1100 and RTR = 1) which means the buffer transmits an ID only as a request for data, and remote response (1010) which is the code to transmit data only on reception of a matching ID from an RTR frame. After transmission or reception of data on the CAN bus, the TouCAN will update the MBs code field and its corresponding bit in the IFLAG register will be set.

Reading the control/status word on a receive MB will lock that MB so that its entire contents may be read without a serial message buffer overwriting any of the data. The MB is unlocked by reading the control/status word of another MB or by reading the 16-bit timer. When polling a MB for completion of data transfer, the IFLAG register should be used and not the code field within the MBs CONTROL/STATUS register as this could lock the MB and prevent a message from being transferred.

The length field specifies how many data bytes are contained within the message – zero to eight. On transmit buffers the CPU writes this value, whereas on receive buffers the TouCAN copies this field from the DLC field within the CAN message.

During reception or transmission of data, the time stamp is captured when the ID field appears on the CAN bus. The captured time stamp value is transferred from the TouCAN's free-running counter to the relevant MB only when the entire message frame has been successfully transferred. If standard format is used, all 16-bits of the time stamp are used, but for extended format only the upper 8-bits of the timer are used.

The ID_HIGH and ID_LOW registers contain the arbitration field just as it appears on the CAN bus, i.e. the SRR, IDE and RTR bits are embedded within the 11 or 29-bit ID. This is why the layout of the bits is slightly different for the standard and extended formats. The user must be careful when writing these registers that the ID value is as intended. The RTR, IDE and SRR bits are as specified in the CAN specification i.e. RTR is set to 1 for remote frame request, IDE is set to 1 for extended format and SRR in extended format should always be set to 1.

The MBs data registers contain up to eight bytes of data as specified by the length field. As with the length field, these registers are written by the CPU and transferred by the TouCAN on transmit buffers, but are written as received from the CAN bus in receive buffers.

The TouCAN uses a double buffering scheme where one of two shadow serial message buffers are employed for temporary buffering of data. This allows the TouCAN to buffer all CAN messages before transferring them to a matching message buffer. These two message buffers are not accessible by the CPU.

4.9.2 TouCAN Bit Timing

The TouCAN bit timing follows the rules defined in Section 3.5. An 8-bit prescaler is used to divide the system clock by up to 256 to obtain a serial clock for TouCAN use. The PRESDIV register is used to select this divide ratio and the resulting clock is called the S-clock, whose period is one time quanta.

The propagation, phase 1 and phase 2 segment values are set by programming the PROPSEG[2:0], PSEG1[2:1] and PSEG2[2:0] bits within CANCTRL1 and CANCTRL2 registers. Since the synchronous segment size is always 1 and the other segments are one more than the value programmed in the register bit field, the bit time length will be:

$$\text{bit time} = 1 + (\text{PROPSEG} + 1) + (\text{PSEG1} + 1) + (\text{PSEG2} + 1) \text{ time quanta}$$

i.e.

$$\text{bit rate} = \frac{F_{SYS}}{(\text{PRESDIV} + 1) \bullet (4 + \text{PROPSEG} + \text{PWSEG1} + \text{PSEG2})} \text{ bits/s}$$

The resynchronization jump width is set by programming the RJW[1:0] field in CANCTRL2 and should not be greater than PSEG1.

For example, if the system clock frequency is 20MHz and the maximum bit rate of 1Mbit/s is required then one possible set-up is as follows:

PRESDIV = 1, i.e. prescaler divide rate is 2 and S-clock is 10MHz, time quanta = 100ns.

PROPSEG[2:0] = 4, so propagation segment is 5 time quanta.

PSEG1[2:0] = 1, so phase 1 segment is 2 time quanta.

PSEG2[2:0] = 1, so phase 2 segment is 2 time quanta.

RJW[2:0] = 1, i.e. resynchronization up to 2 time quanta.

This allows for a propagation time of 5 • 100ns i.e. 500ns which is sufficient for most transmission circuits. RJW cannot be bigger than PSEG1, so is limited to 200ns. On many systems, for instance those with bus lengths less than say 10m, PROPSEG could be made smaller, allowing PSEG1 and PSEG2 and thus RJW to be increased. Larger RJW will compensate for less accurate oscillators or for PLL systems.

There are some restrictions that must be observed when setting the TouCAN bit timing. If the S-clock is equal to Fsys, i.e. PRESDIV = 0, then PSEG2[2:0] must be set to at least 2, otherwise PSEG2[2:0] must be at least 1. The bit time should be at least 9 system clocks in length to guarantee correct operation. This should not pose any problems as in most cases the system clock will be at least 16MHz which means there will always be more than 9 system clocks per bit time since maximum bit rate is 1Mbit/s.

4.9.3 Pin Configuration

RXMODE[1:0] and TXMODE[1:0] in the CANCTRL0 register allow the receive and transmit pins to be configured independently of each other. On the 68376, only the CANRX0 pin is available so the RXMODE1 bit is not used. Clearing RXMODE0 defines a logic '0' on the CANRX0 pin as dominant, which will be the case when using most CAN transceiver ICs. However, on custom designed CAN bus interface circuits, either setting is possible.

On the 68376, only the CANTX0 pin is available and the following settings are possible for TXMODE[1:0]:

00	–	drives '0' for dominant and '1' for recessive.
01	–	drives '1' for dominant and '0' for recessive.
1X	–	drives '1' for dominant and open drain for recessive.

The usual setting when using standard transceiver ICs is 00.

4.9.4 Interrupts

The TouCAN has 19 sources of interrupt, one for each of the 16 MBs, bus off, error and wake up interrupts, and are enabled by setting the relevant bits in the CANMCR (WAKEMSK), CANCTRL0 (BOFFMSK and ERRMSK) and IMASK registers. Each of these sources have individual enable bits and status flags but share common interrupt arbitration and request levels.

The interrupt mechanism for the TouCAN is similar to other IMB modules on the 683xx family. The interrupt arbitration field (IARB[3:0] within CANMCR) may take on values 0 through 15 and is reset to 0. A non-zero value must be assigned otherwise the CPU will process a spurious interrupt.

The interrupt priority level is set using ILCAN[2:0]. 0 disables all TouCAN interrupts, while 7 is the highest priority. The IP field within the CPU status register must be set to a value lower than the ILCAN to enable interrupts of this level. The CPU32 allows nested interrupts – the current

interrupt increases the CPU IP field to a level equal to its own interrupt priority level. This means that any subsequent interrupts of higher level will override the current interrupt whereas lower or equal requests are disabled until afterwards.

Finally, the vector base address is defined for MB0 by writing IVBA[2:0] and the other interrupts will occupy the preceding 18 vector entries in the vector table. The location of the interrupt vectors are calculated by shifting the IVBA value twice, adding the VBR offset and then adding an offset of 4 times the buffer number as shown in the following examples:

IVBA = $40 (vector # 64), VBR = $0000,
MB0 vector address = (IVBA[7:0] << 2) + VBR + (4 x MB#)
= $100 + $0000 + $00 = $0100

IVBA = $60 (vector # 96), VBR = $1000,
MB2 vector address = (IVBA[7:0] << 2) + VBR + (4 x MB#)
= $180 + $1000 + $08 = $1188

4.9.5 Message Filters

There are three 32-bit mask filter registers on the TouCAN which are used to make some of the ID bits within the message buffers ‘don’t care’ bits. The masks are relevant for receive buffers only. RXGMSK provides a global mask for MB0–13. RX14MSK and RX15MSK are used as unique masks for MB14 and MB15. The default state of these registers after reset is all ‘1’s which means that the ID bits within a message must match all the bits in an active receive message buffer before the message is accepted. The user may write any or all of these bits to a ‘0’ to allow any of the ID bits in a message to be ignored. If all the bits in a mask register are set to ‘0’ then an active receive buffer will accept all messages of the correct format (i.e. standard or extended) from the CAN bus. Bit 19 of the mask register corresponds to the IDE bit within the arbitration field and is always a ‘1’, thus a receive buffer cannot be configured to filter both standard and extended formats. If a node has to accept all messages on a bus then a minimum of two message buffers must be used – one for extended and the other for standard frames.

Bits 20 and 0 in the mask register correspond to the SRR and RTR bits in a message. These bits are always ‘0’ and are never compared to the corresponding bit with the received frame.

Although the mask filters allow reception of a data frame with a different ID field to that originally written into the MB’s ID field, the actual ID received on the bus will be copied into the MB. This means the ID within the MB may be altered by a reception

Mask filters are used only for data frames – any remote frames must match the MB’s ID field exactly.

4.9.6 Error Counters

As described in Section 3.4, fault confinement on the CAN bus is achieved using two 8-bit error counters. On the TouCAN, these are the RXECTR and the TXECTR registers. These error counters are automatically increased by the TouCAN by 1 or 8 on detection of Rx or Tx errors and decrement by 1 on completion of successful reception or transmission. As described in Section 3, the node status changes from error active to error passive if either counter exceeds 127 and changes to bus off status if the Tx error counter reaches 256. The status of the node is indicated in the FCS[1:0] bits in the ESTAT register and generation of interrupts may be enabled after detection of an error or a transition to bus off state. The ESTAT register also sets either of two warning flags, TXWARN or RXWARN, if either of the error counters exceed 96 as this indicates a heavily disturbed bus.

4.9.7 TouCAN Initialization

After the TouCAN is reset (power-up, hard reset or soft reset), all of its control registers default to their reset state but the message buffers are not initialized. The TouCAN module will not attempt to communicate with the CAN bus at this stage as its HALT bit within the CANMCR register is set allowing the CPU access to all TouCAN registers. A typical initialization procedure is as follows:

1. Initialize pin configuration and bit timings using the following registers:
   CANCTRL0: RXMODE [1:0], TXMODE[1:0] – define dominant and recessive levels and select CMOS or open drain drive.
   CANCTRL1: SAMP, PROPSEG[2:0] – select 1 or 3 samples per bit and a propagation delay of between 1 & 8 time quanta.
   PRESDIV – select how many CPU clocks (1–256) make up one TouCAN time quanta.
   CANCTRL2: RJW[1:0], PSEG1[2:0], PSEG2[2:0] – select phase segment delays between 1 & 8 time quanta and re-synchronize with up to 4 time quanta. Adhere to rules in bit timing section.
2. Make all 16 message buffers inactive by writing their control field to 0000. The MBs may be activated for Rx/Tx now so they will begin the arbitration process immediately the TouCAN has synchronized with the CAN bus (after step 6) or can be left inactive until the CPU is ready for communication. In either case, the following steps are taken to activate the MB for communication when required:
   Initialize the ID fields of required MBs by writing the ID_HIGH and ID_LOW registers to the necessary values, taking care to embed the SRR, IDE and RTR bits within the 11 or 29-bit identifier. If the MB is to be used for transmit, write the data bytes and length field. Finally, rewrite the control field to make the MBs active Rx (0100) or active Tx (1100), or use one of the remote codes for remote frame set-up.

3. Write the 32-bit ID mask filter registers RXGMSK, RX14MSK and RX15MSK to filter messages with the required ID. As with the ID field in the MB, take care to embed the three non-ID bits within the ID filter. These registers default to ‘1’s in each if the 29 ID bit fields which mean that every ID bit is compared with the ID field in the message buffer. Also, the bit corresponding to the IDE bit (bit 20) is always a ‘1’ which means that IDE cannot be masked and therefor any one MB cannot be configured to receive both standard and extended format frames.

4. Refer to the section on interrupts for a description on interrupt operation. The following registers effect interrupts:
   CANMCR: IARB[3:0] – set interrupt arbitration value to non-zero if interrupts enabled.
   CANICR: ILCAN[2:0] – set between 0 (interrupts disabled) and 7 (highest priority).
   CANICR: IVBA[2:0] – indicates the location of the interrupt vector.
   Finally, select which of the 19 TouCAN interrupt sources are required
   IMASK – write a ‘1’ to each bit corresponding to MBs 0–15 where an interrupt is required on completion of transmission / reception.
   CANMCR: WAKEMSK – set bit if wake-up interrupts are required.
   CANCTRL0: BOFFMSK, ERRMSK – set bits for bus off or error interrupts.

5. Read free running timer to ensure all Rx MBs are unlocked.

6. Finally clear the HALT flag in CANMCR to enable the TouCAN to synchronize with the CAN bus and allow participation in communication.

Example TouCAN software initialization is listed in functions ain_toucan_init and aout_toucan_init in Section 7.

5 Audio Transfer

A system for transmitting stereo audio over the CAN bus using TouCAN is described. The system makes extensive use of the Motorola 68376 microcontroller which combines TouCAN, QADC, QSPI, 7.5K bytes SRAM, CPU32, SIM and other additional modules which are not used in this system i.e. ROM and timers.

The audio input node consists of a stereo input jack for audio input, typically from a personal CD player which connects to two of the QADC analogue input pins. The QADC samples at approximately 60kHz i.e. 30kHz per channel, and the data is passed to the TouCAN which transmits it over the CAN bus via an external transceiver IC.

The audio output node also uses a 68376 with TouCAN receiving the audio data via the CAN bus. The 68376 uses internal SRAM to buffer the incoming data before outputting it on the QSPI to an external serial DAC. The DAC provides two low power output audio signals which are capable of driving a pair of amplified speakers. The hardware design is described in full in Section 6.

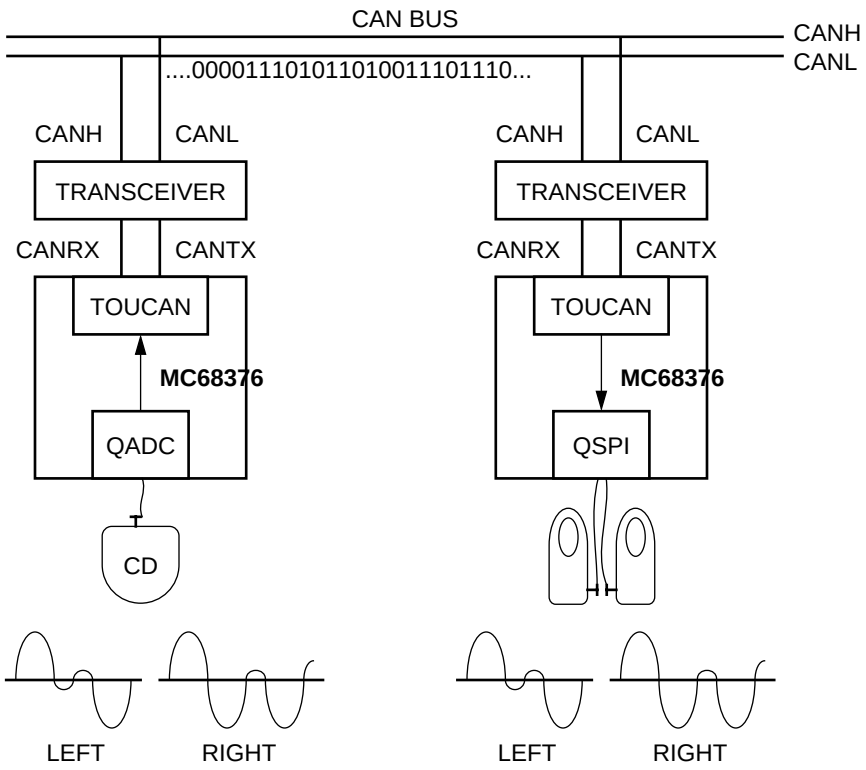

Figure 5 Input and Output Nodes Connected to CAN Bus

5.1 Audio Input

5.1.1 Digital Audio Sampling

The basic task is to take a pair of analogue audio wave-forms and sample them with regular sampling and adequate resolution, or quantization, to allow them to be represented numerically. The numerical digital data will then be reproduced with the minimal acceptable amounts of noise and distortion.

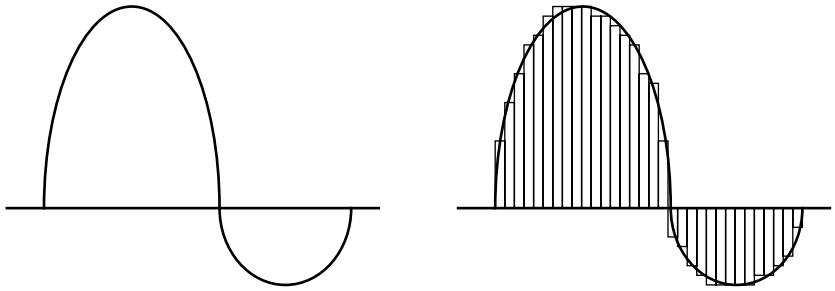

Figure 6 Analogue Wave-Form and Digitized Version

Typical hearing bandwidth starts at around 20Hz and goes up to between 15 or 20kHz. This range narrows with age as the eardrum becomes less flexible. Nyquist's sampling theory states that a signal must be sampled at least twice the rate of it's maximum frequency component if no information is to be lost. This means that for good audio reproduction, we have to sample at a frequency greater than 40kHz, but even down to 30kHz would still capture the audible range detectable by most of us. As a comparison, CD uses a sampling rate of 44.1kHz with 16-bit accuracy (96dB SNR) for audio reproduction up to 20kHz, whilst FM stereo broadcasting and NICAM 728 stereo TV sound system both use a sampling rate of 32kHz with 14-bit accuracy (84dB SNR) for audio up to 15kHz. Voice reproduction on GSM handsets uses a bandwidth of 300Hz to 3.4kHz with 8-bit accuracy (48dB SNR).

Sampling frequencies in this range are well within the capabilities of the 68376's QADC, which can operate at sampling rates above 100kHz with typical conversion times as low as 8.6µs. The main limitation is the CAN bandwidth. Running CAN at 1Mbit/s yields approx. 530kbit/s of data using standard ID format and 460kbit/s using extended ID format due to non-data fields within the message. In our system we will use standard format for higher data bandwidth and assume that most of this is available for audio, but that some additional non-audio messages are present on the bus. In addition, some messages may have to be

retransmitted in the event of an error. Because of these reasons, we will reserve 90% of the available bandwidth for audio transmission. A simple calculation yields maximum sampling rate for each audio channel as follows:

$$\text{max. sampling frequency} = \frac{\text{available CAN bandwidth}}{(2\ \text{channels}) \bullet (\#\ \text{bits in sample})}$$

$$\text{using 10-bit QADC resolution (60dB SNR)} = \frac{530000 \bullet 0.90}{2 \bullet 10} = 24\text{kHz}$$

A sampling rate of 24kHz will capture audio frequencies up to a maximum of 12kHz and is insufficient for good quality playback. We can consider several options which would allow us to increase the sampling rate:

i) combine the stereo inputs into a single mono signal,

ii) use data compression techniques to mathematically compress the data, or

iii) reduce the resolution of the analogue samples.

Option i) is not suitable since our original requirement was for quality *stereo* reproduction. Option ii) is a viable solution but would put additional loading on the CPU. In our system where the transfer of audio data is the sole task of the CPU, this would probably be acceptable, but on other systems where additional tasks may be running, the loading on the CPU may not be acceptable. Besides, digital data compression techniques are out-with the scope of this note. Option iii) is possible – the noise levels introduced using 8-bit sampling still give acceptable audio reproduction. Also, 8-bit data is more suited to the data byte orientation of CAN.

Re-applying our sampling rate calculation with 8-bit quantization gives:

$$\text{max. sampling freq. using 8-bit resolution (48dB SNR)} = \frac{530000 \bullet 0.90}{2 \bullet 8} = 29.8\text{kHz}$$

Our system frequency is not critical and 20MHz has been selected for simplicity. The 68376 is currently available up to 25MHz but this is not required as CAN bandwidth is our limitation, not processing power. The QADC conversion times are based on the QCLK which in turn is derived from the system clock. The minimum QADC conversion time of 18 QCLKs will be used. The prescaler for our QADC clock can now be calculated to give the 29.8kHz sampling rate:

$$\text{QADC prescalar} = \frac{F_{SYS}}{(\text{Samp. freq.} \bullet 2) \bullet \text{Conversion clocks}}$$

$$\frac{20000000}{29800 \bullet 2 \bullet 18} = 18.5$$

A prescaler value of 19 will be used, resulting in a sampling frequency of 29.2kHz, or 342 (19 • 18) system clocks. This results in CAN bandwidth usage of 88%.

To test audio sampling parameters, a non CAN system was developed using a single 68376 with the same QADC input circuitry and QSPI output circuitry as is used on our final system (hardware described in Section 6). This system was used to test hardware, sampling rates and quantization levels before moving onto a more complicated two-node system with CAN. Simple software was used which sampled the audio input and immediately output the data on the QSPI. This system was used to digitally sample a single audio channel from the analogue 'line-out' socket on a personal CD player and playback using an external DAC and amplified speakers. Experimentation on sampling rates and quantization levels demonstrated that 8-bit resolution gave reasonable results with little loss of quality when compared to 10-bit sampling. The improvement gained with 30kHz sampling when compared with 24kHz was much more noticeable and so the final specification of our system shall be as follows:

sampling rate	–	29.2kHz
sampling resolution	–	8 bits
no. of audio channels	–	2
CAN rate	–	1Mbit/s
CAN bandwidth	–	88%

One last subject on digital audio sampling – aliasing. Again, referring to Nyquist's sampling theory and avoiding the detailed mathematics, the sampled baseband signal, Fb, will be reproduced at all harmonics of the sampling rate, Fs, as shown in Figure 6. The original data can be reproduced without loss of data if the sampling frequency is at least twice its highest component.

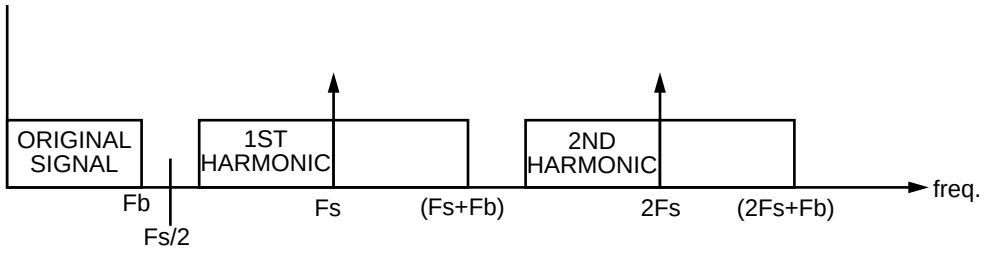

Figure 7a Sampled Signal and Harmonics

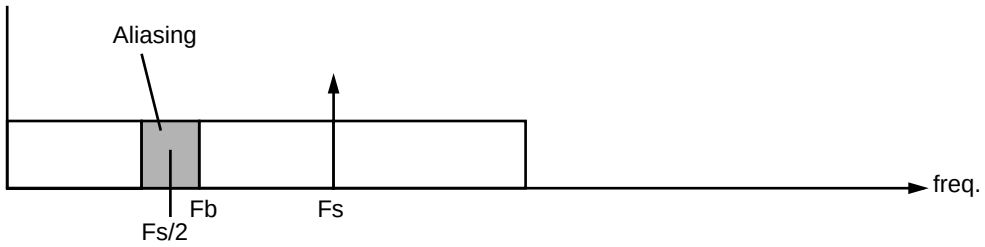

Figure 7b

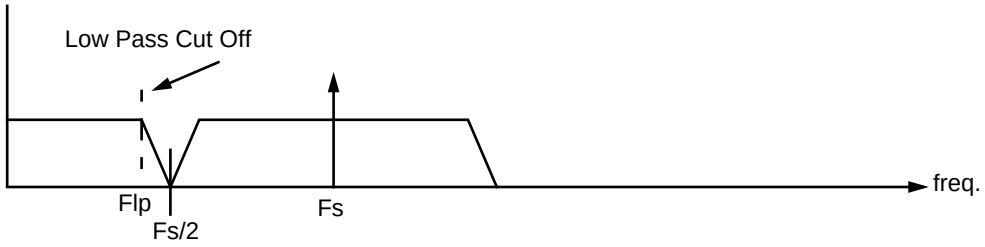

Figure 7c

If the sampling frequency is less than double, as shown in Figure 7b, the reproduced signal at the first harmonic will overlap the original signal and the resulting distortion is called aliasing. To avoid aliasing, the sampling frequency should be increased. If this is not possible, a low pass filter must be used to remove some of the high frequency data within the input signal, as shown in Figure 7c. An expensive LP filter with sharp roll-off allows cut-off frequencies at half Fs. Inexpensive LP filters with gentle roll-off requires the cut-off frequency to be lower, thus even more of the original signal is lost. A trade off between price and quality must be made.

In our system, the input signal from a CD will have 20kHz bandwidth. Sampling at 29.2kHz means that aliasing will exist without the use of an

LP filter. Brief testing on our non-CAN test system using sinusoidal inputs showed that aliasing was indeed present above 15kHz. However, with audio input, the reproduced signal quality was reasonable since audio content above 15kHz is low. Adding a fourth-order LP filter with cut-off frequency around 14kHz in series with the input produced little or no audible difference. For these reasons, and since LP filter design is not a goal of this note, we shall avoid the use of one in our system.

5.1.2 QADC to TouCAN Transfer

The main objective of the microcontroller on the audio input side of the CAN bus is to sample the analogue input using a fixed sampling rate and then transfer the data onto the CAN bus. The QADC will be used in 'software triggered continuous-scan mode' which allows the QADC itself to control the sampling rate to 29.2kHz per channel i.e. 58,400 samples per second. A single queue, queue 2, with maximum length of 40 conversions will be used to reduce the CPU overhead to a minimum. This queue will be configured to sample left audio channel on PQA0 and then right audio channel on PQA1 alternatively. Once all 40 conversions are complete and the queue is full, the QADC will generate an interrupt to request the CPU to empty the conversion result queue and transfer the data to the TouCAN module.

The QADC ISR (interrupt service routine) must move the 40 conversion results immediately as the QADC will loop back to the start of the queue and overwrite the previous results. In particular, the first conversion result must be recovered within 17.1µs i.e. the sampling period, as it is the first to be overwritten during the next iteration of the queue. This process is summarized in Figure 8.

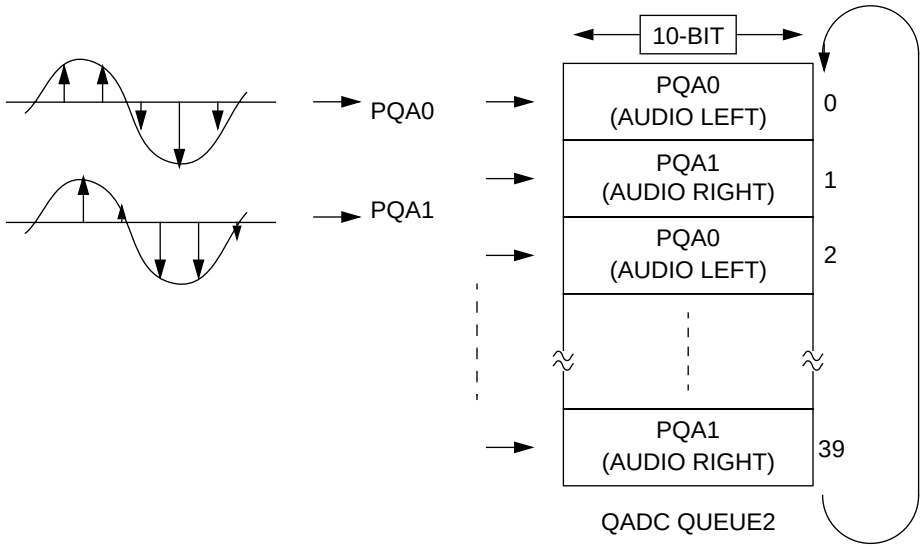

Figure 8 QADC Continuous Sampling Operation

By assigning five of the available sixteen TouCAN message buffers to transmit audio, each with eight data bytes, we can match the 40 bytes from the QADC to the TouCAN data buffer size. The first eight bytes of data will be transmitted from message buffer 0 and the last eight from message buffer 4. Setting the TouCAN to transmit lowest numbered buffer first, rather than lowest ID first (LBUF bit in CANCTRL1), we are assured proper ordering of our sampled data. Once the QADC has completed the conversion queue, all 40 bytes will be transferred to the TouCAN and the five message buffers enabled together. The TouCAN will then commence arbitration for the bus for each of the five messages in turn and requires no further action from the CPU until all five messages, i.e. 40 bytes, have been transferred. The TouCAN setup is detailed in the software section.

As discussed previously, not all of the available CAN bandwidth will be used. This means that on average the CAN will transmit the data faster than the QADC can deliver it. In theory this means that the TouCAN message buffers will always be empty by the time the QADC has completed its queue, allowing the QADC ISR to transfer all 40 data bytes directly from its result queue to the TouCAN. In practice, there may be brief periods of heavy CAN bus activity, either from additional CAN transmitter nodes or due to retransmission of an error-induced message from our own transmitter.

To compensate for brief periods of additional activity, we shall use a small buffer in RAM which may be used as a temporary store for data

from the QADC in the event the TouCAN has not completed its own transfer. A temporary buffer large enough to take the 40 bytes of data from the QADC i.e. five CAN message lengths of eight data bytes each will be sufficient to ensure no audio data is lost as long as there is no more than one additional non-audio CAN message in every ten audio messages. As is shown in Figure 9, when running at 88% bandwidth, one additional non-audio message in every ten will keep us within the available bandwidth limit. Our small buffer will require only simple software control and will protect against data loss when additional but infrequent CAN messages are present on the bus. This mechanism is not suitable when additional data of two or more consecutive messages with high priority appear on the bus. In this situation, all 40 bytes in the buffer would be overwritten by the QADC ISR before the TouCAN could retrieve them and therefor all 40 bytes would be lost. This situation could be avoided by assigning highest priority identifier to the audio data. In our system, the audio data shall be assigned a high priority ID for this reason. In addition, each of the five audio messages from the five message buffers shall be assigned different identifiers. This will give us better visibility in a debug environment and also allows demonstration of the receiver filter mask on the audio output node. The 11-bit ID for message buffer 0 shall be set at 000 i.e. highest possible priority, through to an ID of 004 for message buffer 4. A receive mask shall be used on the receiving side of the bus to filter off the three least significant bits of the ID, thus all five audio message may be received using a single receive ID plus one of the three available 32-bit filter registers. See Section 5.2.5 for explanation of receive filter.

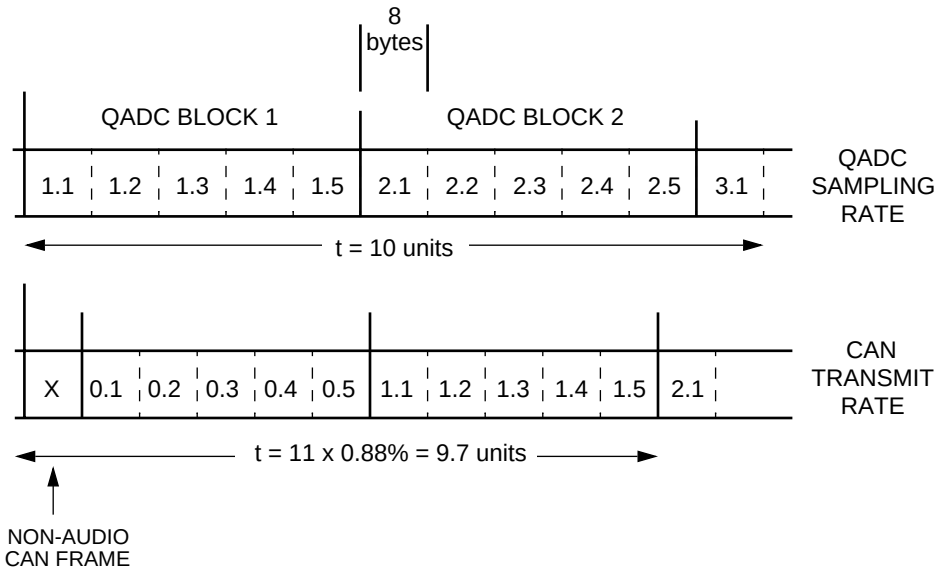

Figure 9 Additional Loading on the CAN Bus

The flowcharts for the QADC and TouCAN ISRs are shown in Figure 10 and Figure 11 respectively. The control of data flow from QADC into either TouCAN or RAM buffer is handled by two flags – CAN_BUSY and RAM_FULL as shown. These two routines must be mutually exclusive and this is easily accomplished by assigning both ISRs the same interrupt request level. The TouCAN interrupt should be triggered when all five message buffers are empty, so only the last message buffer, MB4 will be used for interrupt generation.

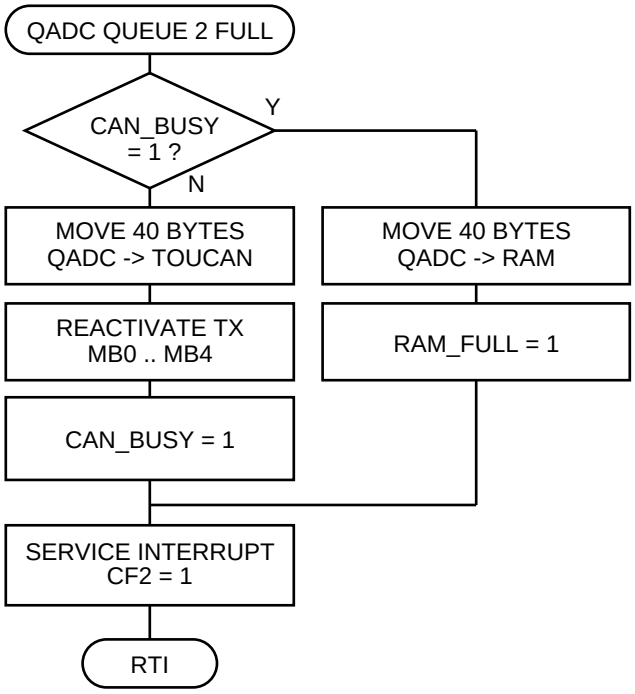

Figure 10 QADC ISR Flowchart

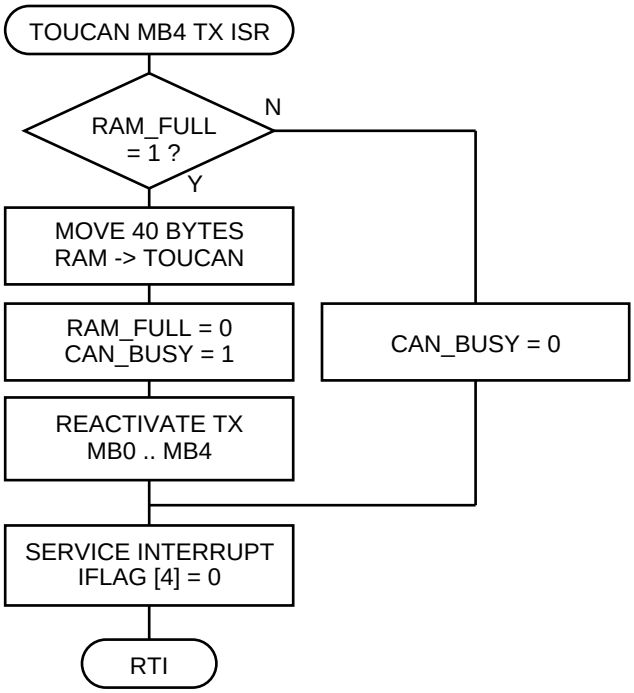

Figure 11 TouCAN Tx ISR Flowchart

These two ISRs, along with the QADC's queued mechanism and the TouCAN's multiple message buffer structure, completely handle the audio input function of sampling the dual analogue audio wave-forms to transmitting the numerical representation onto the CAN bus. This means that this function is fully interrupt driven and in our system, the main audio input routine becomes a 'do-nothing' loop after performing initialization. In other systems this is significant if audio transfer has to be performed as a background task.

5.2 Audio Output

The node at the audio output has to receive all data on the CAN bus and filter off non-audio information. It must then separate left and right channel information and reproduce the original stereo audio signals.

5.2.1 Digital to Analogue Converter

To generate the analogue wave-form, an external DAC (Digital to Analogue Converter) IC will be interfaced to the 68376. Serial input DACs are abundant and are less expensive and come in smaller pin packages than a corresponding parallel input DAC. The QSPI offers an ideal interface for a serial DAC and may be configured for most, if not all available. Our selection criteria for a DAC is as follows:

- serial interface for QSPI connection
- dual channel to reproduce left and right stereo
- minimum 8-bit resolution – our system currently uses 8-bit sampling, but all 10-bits of the QADC resolution may be implemented in a future revision if CAN bandwidth is better utilized, i.e. by making use of data compression techniques.
- single +5 volt supply
- voltage output drive capable of driving a pair of amplified multimedia speakers

The MAX549 DAC was considered. This is a low cost, dual 8-bit, voltage output, serial input device in 8-pin DIP package. The drawback with this device is its high output impedance and limited 8-bit resolution.

The AD1866 DAC is a more expensive device and comes in a 16-pin package but offers all the features required. It is a dual 16-bit, voltage output, serial input device running from a single +5 volt supply. In addition, this DAC is intended for audio applications and has the advantage of offering audio compatible outputs i.e. $\pm$ 1 volt up to $\pm$ 1mA. This DAC is the one chosen for our application and is examined in more detail in the hardware sections later.

This device requires the serial input to be supplied as a 16-bit serial stream with two's compliment, MSB first format. The QADC offers the digital conversions in three formats – the Left Justified, Signed Result

Register or ‘LJSRR’ is the format suitable in this case and will be used on the audio input node.

5.2.2 QSPI Operation

The queuing mechanism on the QSPI will be fully utilized on the output node just as it was on the QADC at the input node. The QSPI will initiate all serial transfers to the DAC, thus will operate in master mode. All 16 command queue entries will be used to perform 16 serial transfers without CPU intervention. In addition, wrap-around mode will be enabled to allow continuous execution of the queued commands. The QSPI output timing will be set to match the input sampling rate of the QADC. This greatly reduces the overhead on the CPU as it does not have to generate the continuous output of data at approximately 58,400 bytes per second – this will be a task for the QSPI. The CPU merely has to ensure that there is always updated audio data within the 16 word queue.

5.2.3 QSPI Data Output Timing

Data has to be serially shifted out at the same continuous rate as the QADC sampling frequency, i.e. 58.4kHz (29.2kHz per channel) i.e. every 342 system clocks. Each serial transfer from the QSPI consists of three components – delay before transfer, the serial transfer itself and a delay after transfer, as shown in Figure 12.

Since the AD1866 DAC requires a fixed 16-bit transfer, the 8-bit digital word must be zero extended. Maximum QSPI operating shift rate of Fsys/4 will be used, thus the number of system clocks for the actual 16-bit shift will be 4 • 16 i.e. 64 system clocks. The delay before transfer is adjustable by single system clocks in the range 1 to 127. The delay after transfer is adjustable in steps of 32 system clocks from 32 to 8192. Choosing delay before of 22 and delay after of 256 results in a total transfer time of 342 system clocks which matches the sampling rate of the QADC.

word serial transfer time	=	delay before + transfer time + delay after	
	=	22 + 16 • 4 + 256	= 342 system clocks
transfer frequency	=	20MHz / 342	= 58.2 kHz

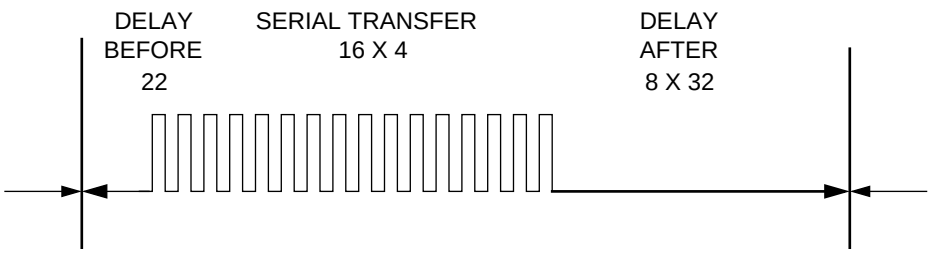

Figure 12 QSPI Serial Data Transfer

This setup is performed in function aout_SPI_init in file aout.c in the software section.

5.2.4 QSPI Data Updating Mechanism

Once all 16 command entries in the queue have been executed, the QSPI Finished Flag, SPIF, is asserted. In wrap-around mode, the next data word at the start of the queue is latched into the shift register for transmission almost immediately after the SPIF flag is asserted. For this reason, we cannot simply wait for SPIF then reload 16 new data entries as the QSPI would already have latched old data for transmission from the first word in the queue. The SPIF flag is more suited to serial reception of data when used in wrap-around mode rather than transmission.

To get around this issue, we will load data into the queue in two halves. Referring to Figure 13, when the queue pointer passes the mid-point of the table, i.e. the first eight queued words have been transmitted (CPTQP = 7), the CPU will load eight new data words to the top of the queue. When the QSPI transmits the last word in the queue and SPIF triggers, the CPU will load eight new data words to the bottom of the queue. This two stage loading mechanism allows the QSPI to run continuously since data will be loaded into the queue approximately eight transmit periods before they are due to be transmitted. The CPU latency time after CPTQP = 7 and SPIF is eight transmit times at 58.4kHz per word i.e.

$$\text{CPU latency for QSPI load} = \frac{8}{58400} = 137\mu s$$

Since only SPIF can generate an interrupt and not CPTQP = 7, we shall not use interrupts for the QSPI. Instead we shall poll for both of these conditions before loading new data into the QSPI transmit data queue. In another system where polling was not possible, SPIF interrupts could

be used to capture the end of queue condition and a timer interrupt could be used to interrupt just after the mid-queue condition.

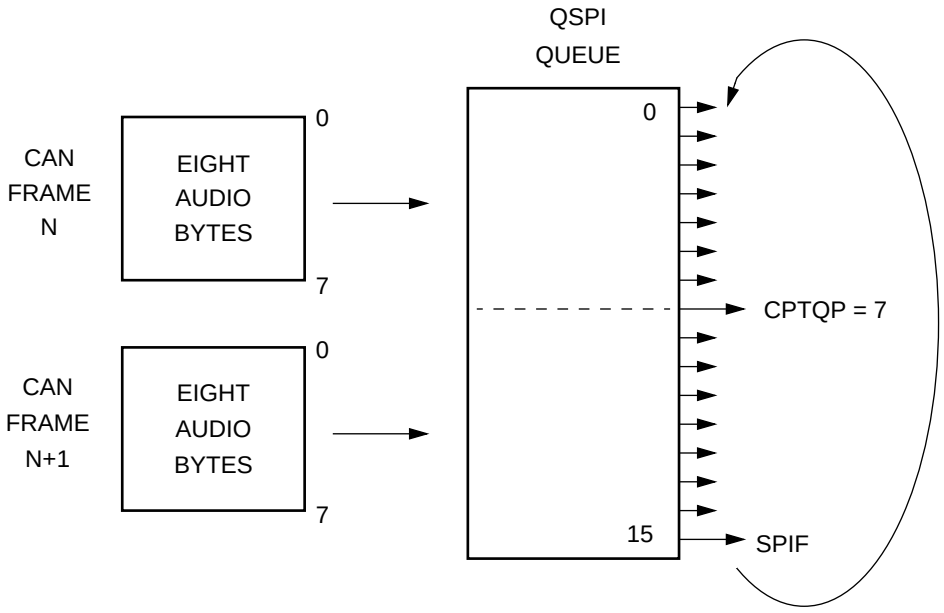

Figure 13 QSPI Queue Mechanism

The QSPI updating mechanism is shown in Figure 16.

5.2.5 Receiving Data from CAN

Receiving the audio data from the CAN bus will be a relatively straight forward process. Although five message buffers are used to transmit audio onto the CAN, this was required only to match the data buffer size of 40 bytes to that of the QADC result buffer. Only one message buffer is required to receive the data from the CAN. This message buffer will use one of the three available mask filter registers to exclude non-audio messages.

Any one of the sixteen available message buffers may be chosen. For our system we will use MB6 which shares the RXGMSK (Receive Global Mask Register) with MB0 – MB13. This shall be set to 0xFF0FFFFE i.e. mask out the three bits (MD18–20 correspond to the first three ID bits in standard format) that differ in the five audio IDs so that all five will pass the filter process.

Receive interrupt will be enabled for MB6 by setting bit-6 in the IMASK register. The resulting ISR must take the eight data bytes from MB6 and make them available for the QSPI. As explained previously, we cannot write directly to the QSPI data queue as this can only be done at certain

instances. For this reason, MB6 ISR must move the eight data bytes into a temporary RAM buffer. The source code for this ISR is listed in aout_MB6_ISR in file aout.c and is further explained in the software section.

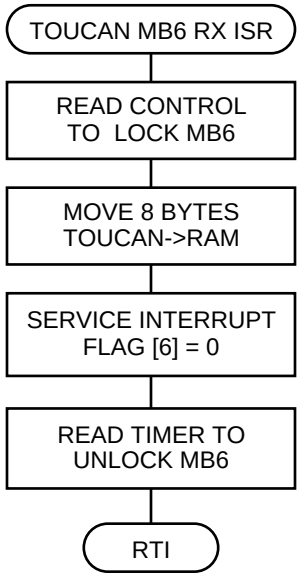

Figure 14 TouCAN Rx ISR Flowchart

5.2.6 Synchronizing Audio Output to Input

Since two micro-processors using separate crystal oscillators are used, it is unavoidable that the data output rate will differ slightly from the input sampling rate. Although this slight difference in speed cannot be detected by ear, it will result in gaps in the audio sound that will be audible. If the output rate is slower than the input, then the RAM buffer between TouCAN and QSPI will eventually fill. When this happens, the newly received data must be discarded as there is nowhere to store it. Since the CAN message consists of eight data bytes, data will be lost in blocks of eight bytes at a time. This is equivalent to an audio transmission of 137 µs. By discarding this data, the output buffer will recover for a short period then overflow again. Similarly, if the output rate is faster than the input, there will be periods when the QSPI data queue has not been updated with fresh data. Since the QSPI is running in wrap-around mode, it will continue to output data regardless and so will repeat previously transmitted data.

Typical crystal accuracy's are in the region of ± 30 ppm. In addition, many 68376 systems make use of the internal PLL (phase-locked loop). The long term clock jitter of the resulting clock generated by the 68376 PLL is listed as 0.0625% with 'long term' defined as 500 µs. In practice the resulting clock will be much more accurate than this figure, but we

shall use these figures for calculating the worst case rate of data loss as follows

$$\text{worst case data loss rate} = 2 \bullet (\text{PLL variation} + \text{crystal variation})$$

$$= 2 \bullet \left(\frac{0.0625}{100} + 30 \bullet 10^{-6} \right) = 1310 \text{ppm or } 0.131\%$$

At a data rate of 58.4kHz, this equates to a maximum loss of 76 bytes per second.

This loss of data will produce small gaps in the output transmission and would certainly be detected by ear. The effect will depend on how large the variation between the two clock rates actually is, but would manifest itself as noise on the audio signal and must be avoided. The question is how do we synchronize output to input and thus avoid any loss of data? One answer may be to avoid the use of the PLL and select a highly accurate crystal oscillator. This would add cost to the system and would still result in slight differences between the two clock rates.

Another suggestion may be to use one external oscillator and share the clock between both processors. In most systems using CAN bus to communicate between remote nodes, transmission of a 20 MHz clock signal would not be practical.

A final suggestion may be to have the audio output node dynamically adjust the output rate of the QSPI so that there is always new data available within its RAM buffer. This may be achieved without any additional hardware cost other than that of the RAM used to buffer the data at the output node. Our system already has 7.5K bytes of SRAM internal to the 68376 and so we will make use of this technique.

The RAM buffer used is shown in Figure 15. This buffer must be a circular buffer i.e. there is no beginning or end, but instead is a loop where data is continuously pushed in at the top and pulled from the bottom. The TouCAN ISR routine as discussed in the previous section, will take eight data bytes from the receive message buffer MB6 and place them onto the circular RAM buffer at a position defined by a

'data-in' pointer. The CPU will then transfer them into the top or bottom half of the QSPI data queue.

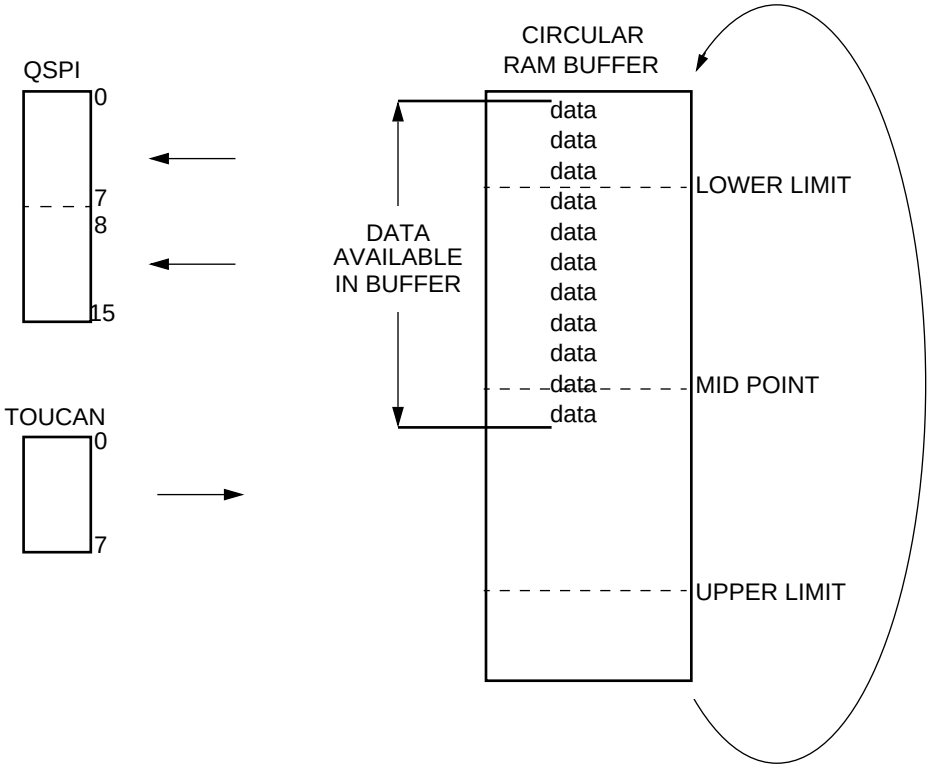

Figure 15 Audio Output Data Buffer

The function of the QSPI control loop is to move data from the RAM buffer to the QSPI and monitor the amount of data within the circular buffer i.e. the distance between the data-in and data-out pointer. The QSPI output rate will then be adjusted so that the amount of data in the buffer is less than an upper limit to avoid having to discard data from the TouCAN, but greater than a lower limit to avoid QSPI being starved of data. An algorithm will be used where three possible QSPI data rates may be selected as follows:

NORMAL – the nominal QSPI speed will be that previously calculated i.e. 342 system clock periods. This will be selected when the circular buffer data size reaches mid-way.

FAST – the fast speed will be selected when the circular buffer data size is greater than the upper limit to allow the QSPI to empty the data at a more rapid rate and will be 341 system clocks.

SLOW – the slow speed will be selected when the circular buffer data size is less than the lower limit to allow the TouCAN to build the data stack upwards and avoid a zero data situation and will be 343 system clocks.

These small variations are approximately 0.3% of NORMAL and will do little to effect the quality of our audio output.

The operation of the QSPI on the circular buffer can now be analyzed. After system reset, data size will be zero i.e. less than the lower limit and the QSPI SLOW rate will come into effect. The SLOW speed will be such that the QSPI removes data at a slower rate than the TouCAN places incoming data onto the buffer. The data size will slowly increase until it reaches the mid-point. Now the QSPI NORMAL rate will be selected to restrict the increase of data in the buffer. If the two nodes have identical clock rates, the buffer will remain centered at the mid-point and the NORMAL rate will remain. However, if the input clock is faster than the output, the buffer content will continue to increase, although at a slower rate than before. Eventually the upper limit will be reached and the FAST rate will be selected which will lead to a reduction in buffer content until once again the mid-point is reached. This will result in a continual switching of QSPI output rate between NORMAL and FAST and the buffer size fluctuating between mid-way and the upper limit. This adjustment on audio output rate is very small and will not be audible.

A similar description may be applied when the input clock is slower than the output. Here the QSPI will toggle between NORMAL and SLOW and the buffer size will fluctuate between midway and the lower limit.

5.2.7 Altering QSPI Output Rate and Buffer Size

The minimum adjustment of QSPI output rate is one system clock and can be achieved by adjusting the DSCKL field i.e. the delay before transfer. Since nominal speed is 342 system clocks, adjustment by one clock period to either 341 or 343 gives a QSPI output frequency variation of approximately 0.3%. This is sufficient to override the maximum possible clock difference between the input and output nodes which was shown to be 0.131%.

Choosing a buffer size of 800 bytes will allow us to buffer up to 100 CAN messages of eight data bytes each. As explained previously, the ‘data-in’ pointer will traverse between either lower limit and mid-point or mid-point and upper limit i.e. QSPI speed will alter after approximately 400 data bytes have accumulated. Since the output node may be running the QSPI at up to 0.3% variation from the input sampling rate, the minimum time for speed alteration may be calculated:

$$\text{min. QSPI output rate adjustment period} = \frac{\text{buffer size}}{\text{max. data accumulation rate}}$$

$$\frac{0.5 \bullet 800 \text{ bytes}}{58.4\text{K bytes/s} \bullet 0.3\%} = 2.3 \text{ seconds}$$

Therefor choosing a buffer size of 800 bytes can cause the QSPI output rate to alter as often as every 2.3 seconds. A smaller buffer may be used i.e. a buffer of 96 bytes would still allow buffering of up to ± 6 data frames of 8 bytes each. On our system using a 68376 with 7.5K bytes of RAM, 800 bytes is not a problem.

It is not possible to adjust the QSPI output rate at any time. The DSCKL field is within the SPCR1 register which cannot be altered while the QSPI is running or operation will be disrupted. Our algorithm for altering speed will involve the following steps:

i) clear WREN bit in SPCR2 to disable wrap-around mode so that QSPI output ceases once last word in queue is sent

ii) wait for transmission of last word by monitoring SPIF in the SPSR (QSPI status register).

iii) re-enable wrap around mode by setting WREN bit.

iv) write new value to DSCKL and start continuous transmission by setting SPE, both within SPCR1 register.

These steps must be completed as quickly as possible so that no interruption in audio output is detected. In our software the QSPI is paused only for two or three instructions i.e. approximately one micro-second thus the change over is almost unobservable.

The QSPI updating mechanism is shown in Figure 16 and is coded in file aout.c.

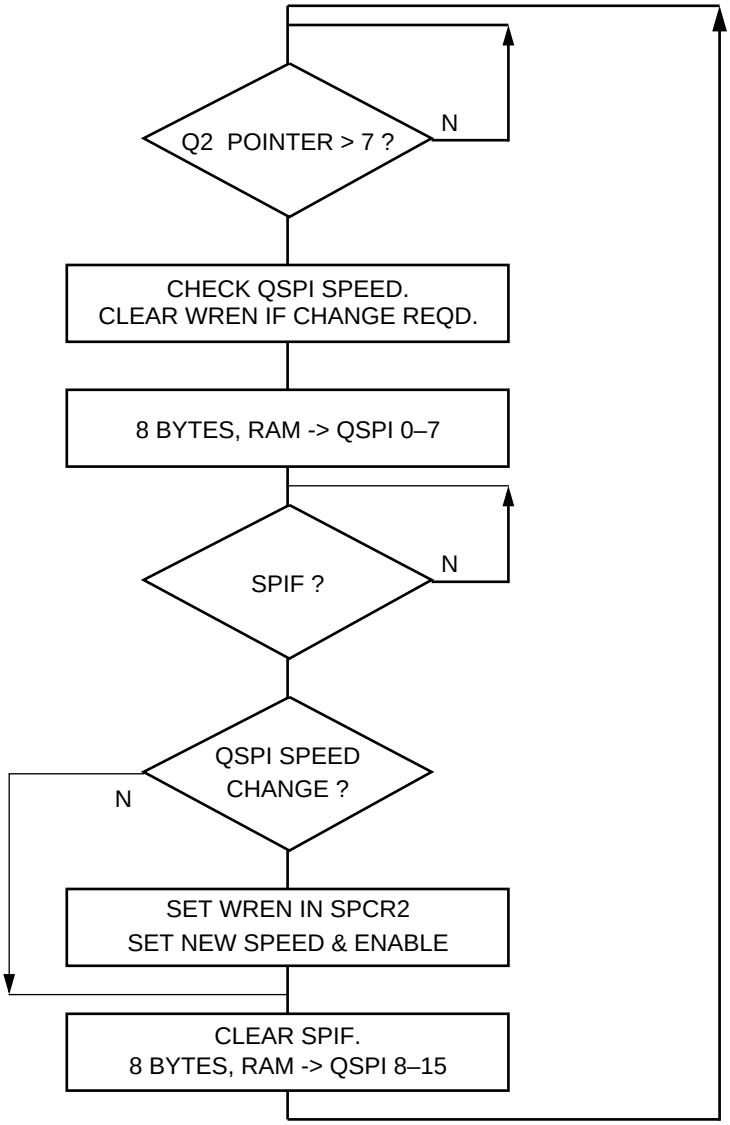

Figure 16 Audio Output Main Routine

6 Hardware Design

The audio hardware consists of biasing and filtering of the analogue audio signal from the audio input (i.e. CD) to the QADC and on the output node, a DAC interface from the QSPI to a set of amplified speakers. The CAN hardware consists of a CAN transceiver IC on both nodes and bus termination resistors at either end of the transmission line.

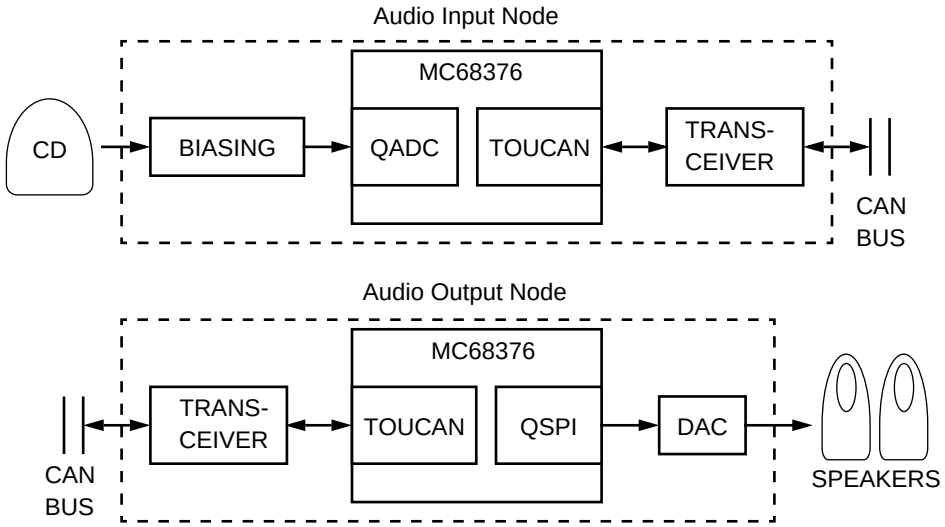

Figure 17 Hardware Circuitry

6.1 Audio Input Hardware

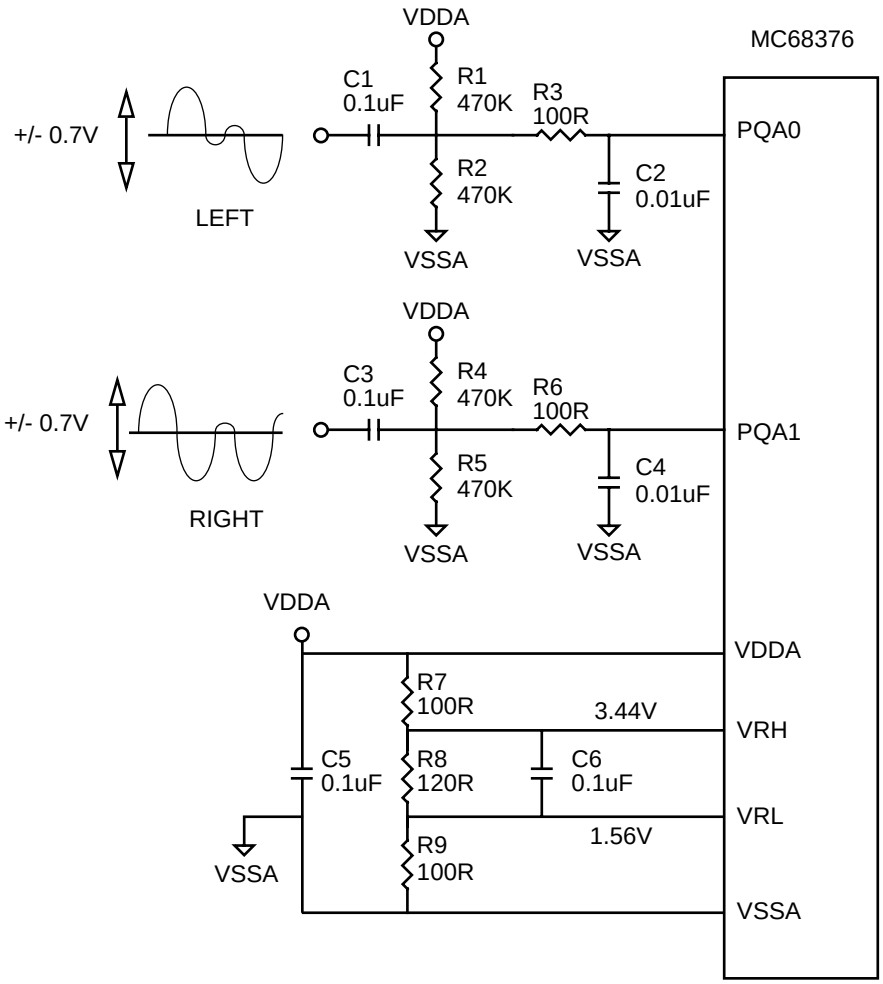

Figure 18 Audio In Hardware

The audio output signal from the line out jack on a personal CD player is typically ± 0.7V maximum. The open-circuit output voltage from the headphone socket may be as high as ± 1 volt. In both cases, the voltage is centered around 0 volts.

Referring to Figure 18, C1, R1 and R2 provide dc biasing for the audio input signal and center it around Vdd/2 i.e. 2.5V. This circuit is also a high pass filter with cutoff frequency of $1/2\pi RC$ where R = R1 in parallel with R2. Choosing large values for R1 and R2 prevents the lower audible frequencies being cut off. The cutoff frequency for our component values ($C = 0.1\mu F$, $R = 235k\Omega$) is 7Hz, allowing the audible frequency of 20Hz to pass.

The second stage of the audio input circuit comprising R3 and C2 is a low pass circuit as recommended for the QADC analogue pins. This low pass filter is designed to filter out any high frequency noise from external digital circuitry or clocks. This particular circuit uses smaller values for R and C than is usually recommended so that cutoff frequency is high enough to prevent attenuation in the audio frequency band. Using R3 = 100Ω and C2 = 0.01µF results in a cutoff frequency of 160 kHz which is high enough to pass upper audio band of 20kHz, yet low enough to filter most of the digital noise which on our system will be around 20 MHz from the system clock.

With ± 0.7V input centered at Vdd/2, the signal range into the PQA0 and PQA1 pins will be from 1.8 to 3.2 volts. To achieve full-scale, full-range results from the QADC, this input range should match the reference voltage range on the VRL / VRH pins. We can do this using one of two methods:

1. amplify the input voltage range by a factor of approximately three so the input range is almost 5 volts, or
2. set VRL and VRH to match the audio input voltage range.

Both these methods have their relative merits. The first requires additional circuitry i.e. non-inverting amplifier, but would help reduce the error due to noise assuming we could amplify the audio signal before injection of noise from the digital circuitry. The second method requires simple circuitry, i.e. only a simple potential divider circuit to bias VRH and VRL. The disadvantage here is that VRH and VRL are below the minimum specified reference voltage differential range for the QADC i.e. 4.5 volts. The QADC will still function using our reduced reference voltage range, but full accuracy is not guaranteed over the smaller range.

To simplify hardware design we will choose option 2) and reduce the V_{REF} differential range to match the audio input range. Any loss of accuracy in the analogue conversions caused by the QADC operating outside its guaranteed range is not critical in an audio system, especially not ours where we have discarded the two LSBs of the conversion. Choosing values of 100Ω, 120Ω and 100Ω for R7, R8 and R9 respectively gives VRL of 1.56V and VRH of 3.44V which sets the QADC close to full scale over the non-amplified audio input voltage range. Finally, C5 and C6 provide capacitive bypassing to help reduce noise on the analogue supply and A/D reference voltage.

6.2 Audio Output Hardware

For reasons described in Section 5.2 Audio Output, the audio output hardware will consist of the AD1866N dual 16-bit audio DAC and the 68376's QSPI synchronous serial interface port. The hardware connections are shown in Figure 19.

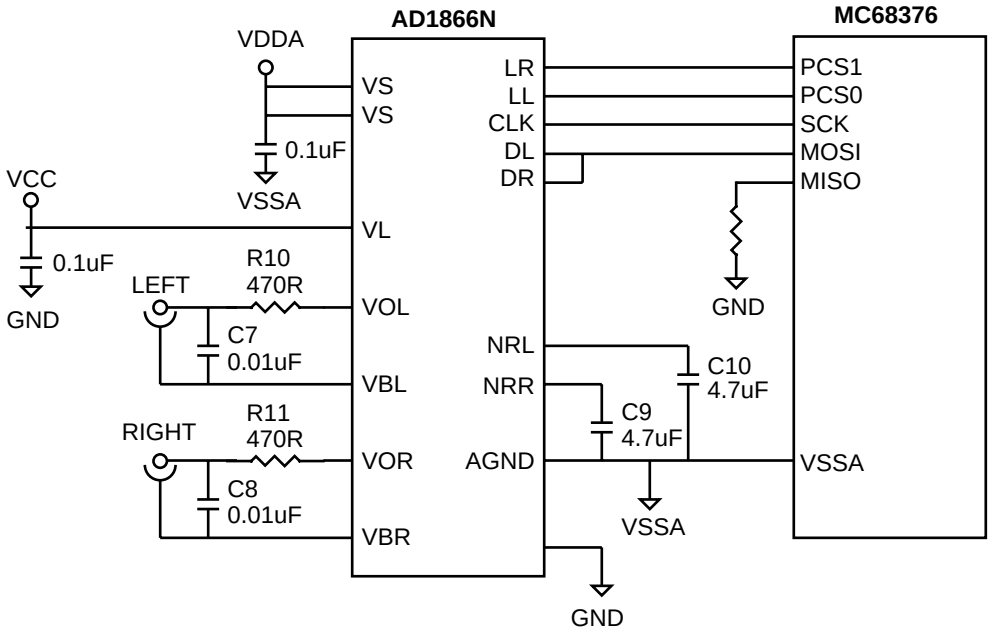

Figure 19 Audio Out Hardware

6.2.1 Digital Signals

Since the QSPI will be operating in master mode, it supplies the serial transfer clock (SCK) and the data output (MOSI). The MISO pin is not required since there is no incoming data for the QSPI. The QSPI will supply data for both the left and right channels from the MOSI pin, so the DAC left and right data input pins, DL and DR, are tied together. Two programmable chip selects, PCS0 and PCS1, are used to control the DAC's left and right channel latch enables and will assert on alternative 16-bit data transitions. The data transfer is summarized below.

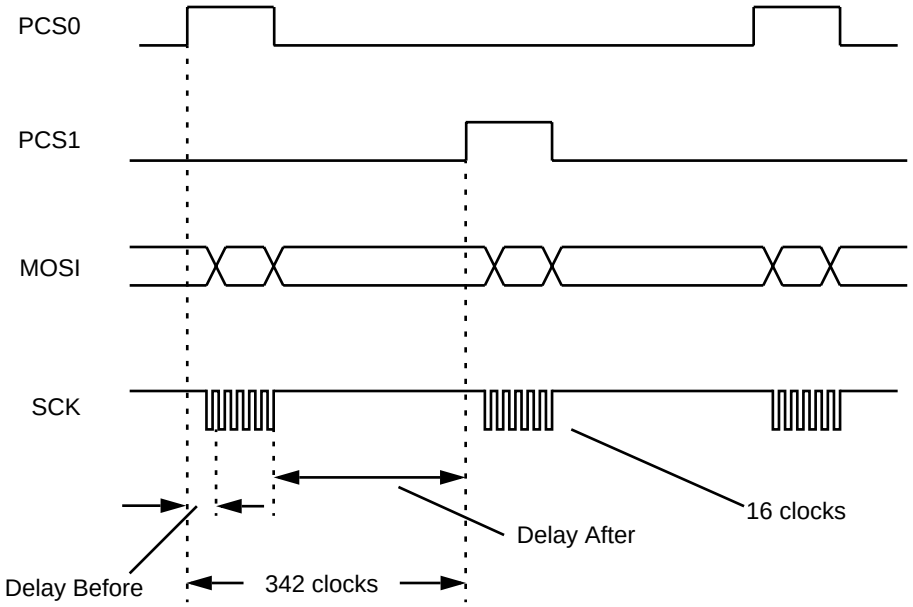

Figure 20 Audio Output Data Transfer

6.2.2 Analogue Signals

The left and right audio analogue signals are driven from VOL and VOR respectively. VBL and VBR supply a dc bias equal to the center of the output voltage swing providing dc coupling to a pair of amplified speakers. Simple first order low pass filters comprising R10/C7 and R11/C8 with a cut-off frequency of 38 kHz is all that is required to refine the quantized edges – the high frequency harmonics of the sampled signal are outwith the human hearing range anyway. The output voltage range from the DAC is ± 1 volt and will directly drive a pair of amplified personal CD or multimedia speakers with input impedance above 1kΩ.

The AD1866 has two separate power supplies to limit the effect of digital noise on the analogue signals. VL and DGND are connected to the digital supply and VS and AGND should be connected to a separate power supply if available. If no separate analogue supply is available then a good PCB layout would provide separate traces from the main power input to the digital and analogue portions of the circuitry.

Finally, capacitors C9 and C10 reduce the output noise contributed by the DAC's internal voltage reference circuitry.

6.3 CAN Hardware

Figure 21 shows the TouCAN interface to the physical CAN bus using an 82C250 transceiver IC. The TouCAN CANRX0 and CANTX0 pins are connected directly to TX and RX on the transceiver. The CANH and CANL signals connect directly to the physical bus. The CAN bus is typically a two wire twisted pair, possibly shielded and must be terminated with two 120Ω resistors – one at each end of the bus. V_{REF} is tied to ground to select the high speed operating mode i.e. no slope control.

A high or floating signal input from the TouCAN to the TX pin forces the transceiver into dual transmit and receive mode. It's CANH and CANL bus pins float to approx. equal voltages around 2.5V (recessive). If no other node drives the CAN bus then the RX output drives high to indicate the bus is recessive. If another node drives a dominant level on the bus, the transceiver drives RX low. If the TouCAN was transmitting at this instant, then this usually means that it has lost arbitration of the bus to the node that transmitted the dominant bit.

A low signal input to the TX pin from the TouCAN forces the transceiver into transmit mode. Its CANH and CANL pins drive the dominant state onto the bus. CANH will be approximately 3.5V and CANL approximately 1.5V i.e. 2V differential. This condition forces RX low which echo's the dominant state back to the TouCAN.

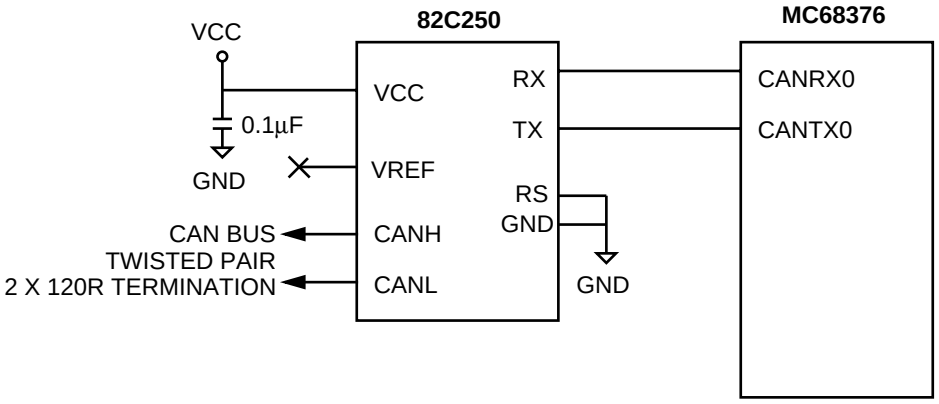

Figure 21 CAN Bus Interface

7 Software

This section describes the software required for the TouCAN, QADC and QSPI set-up for control of the audio transfer described previously and includes source code in C. The software comprises three C source code files plus three header files. General MCU initialisation is not provided – it is assumed that the system clock has been set to 20MHz and the internal software watchdog has been disabled.

7.1 File Summary

ain.c – this is the code which will run on the audio input node and provides functionality to sample audio into the system using two QADC channels and transfer the data out on the CAN bus.

aout.c – code for the audio output node will receive data from the CAN, buffer it in SRAM and then transfer it on the QSPI to an external DAC where it may be replayed on a pair of amplified speakers.

toucan.c – this file contains functions that are common to both audio input and output nodes and will be included in the linking process for both object files.

audio.h – general defines for constants in ain.c and aout.c.

regs.h – this header file contains the declarations for the subset of 68376 registers used plus type definitions.

toucan1.h – constants and bit field definitions used in the toucan.c file.

7.2 Generic TouCAN Routines – file 'toucan.c'

There are four generic routines in the file 'toucan.c' – three provide basic TouCAN functionality and one allows alteration of the interrupt mask in the CPU status register. These functions are common to both the audio input and audio output nodes and will be linked with ain.c and aout.c respectively to form object files for the input and output nodes. The functions are as follows:

7.2.1 toucan_init_global

All general TouCAN initialization is performed by writing to the CANMCR, CANCTRL0, CANCTRL1, PRESDIV, CANCTRL2 and CANICR registers. The TouCAN module is left in HALT mode and the 16 message buffers are left in an inactive state. Instead of passing a lengthy string of arguments to this function, a table of constants is defined in the header file 'toucan1.h'. These constants define all the parameters that are usually fixed at design time i.e. bit timing, etc. and are used for all nodes on the bus.

7.2.2 toucan_MB_off

TouCAN message buffers will reset to an undetermined state after power-up and must be initialized before enabling the TouCAN module. This routine resets all 16 buffers to an inactive state by writing the code field in the control/status registers. This function also clears the IFLAG register to zero for completeness.

7.2.3 toucan_bus_on

This function is called to commence TouCAN activation on the CAN bus after all registers and message buffers have been initialized as required. The free running timer is read to unlock all receive buffers and the HALT flag in the MCR is negated to start TouCAN synchronization with the external bus.

7.2.4 set_ip

This function is included in this file as it is generic to both audio in and audio out. It enables maskable interrupts by writing the required IRQ level to the IP mask within the CPU32 Status Register. An assembler instruction is required to access the Status Register.

7.3 Audio Input – file 'ain.c'

File 'ain.c' is linked with 'toucan.c' described above to form the object code for the audio input node. Referring back to Section 5.1, the audio input functionality is provided by the two interrupt service routines as shown in Figure 10 and Figure 11 – one for the QADC and one for the TouCAN. These two ISRs plus their initialization are formed by four functions in ain.c as follows:

7.3.1 ain_toucan_init

This function initializes the TouCAN for transmission with MBs 0–4. The three generic TouCAN routines are used here. The following operations are performed:

- Initialize TouCAN global registers including bit timing and pin configuration.
- Set interrupt request and arbitration fields.
- Set MB0–MB4 as transmit buffers, each with eight data bytes i.e. 40 data bytes total to match QADC queue size.
- Enable interrupts for MB4 transfer completion.
- Enable TouCAN for CAN bus participation.

7.3.2 ain_QADC_init

Here, the QADC is initialized to perform audio sampling at the required frequency on the two audio channels. The following parameters are set:

- Continuous conversions on QADC queue 2 using a 40-word conversion queue and enable interrupts on queue completion.

58.48kHz sampling rate (342 system clocks) using two alternating input channels from left and right audio inputs i.e. PQA0 and PQA1.

7.3.3 ain_Q2_ISR

This is the interrupt service routine for QADC queue 2 complete as shown in Figure 10. Software loops are avoided here to reduce the ISR execution time at the expense of slightly larger code size. This routine transfers the 40 conversion results to either the TouCAN or a temporary buffer in SRAM.

7.3.4 ain_MB4_ISR

This is the interrupt service routine for TouCAN MB4 transmission complete as shown in flowchart Figure 11. This interrupt indicates that MB0–MB4 have completed transmission and are ready for more data. If data is available in the SRAM then it is transferred into MB0–MB4 otherwise the ISR is exited and the QADC ISR will refill the MBs a short time later. The interrupt is serviced by clearing the relevant bits in the TouCAN’s IFLAG register.

7.3.5 Audio-in main routine

The CPU idles in a ‘do nothing loop’ awaiting either of the ISRs. In another system, the CPU may perform additional tasks here.

7.4 Audio Output – file ‘aout.c’

File aout.c plus the generic routines in toucan.h are linked together to form the object code for the audio output node. This functionality is described in Section 5.2. The main loop polls the QSPI and may make slight adjustments to the QSPI transfer rate to control the size of the data queue in the SRAM buffer due to system frequency variations in the two nodes. The reception of data on the CAN bus and subsequent transfer into the SRAM buffer is performed by the TouCAN interrupt routine. The following functions are provided:

7.4.1 aout_toucan_init

Initialize TouCAN global registers including bit timing and pin configuration.

Set interrupt request and arbitration fields.

Set MB6 and the Receive Global Mask Register to receive all audio data frames

Enable interrupts for MB6 data reception.

Enable TouCAN for CAN bus participation.

7.4.2 aout_SPI_init

This function initializes the QSPI so it may output the audio data to an external DAC for replay. Enable QSPI as master with 16-word queue and wrap-around transfer. Set alternate transfers to activate either audio left or audio right chip selects i.e. PCS0 and PCS1. Start QSPI transmitting at the nominal transfer rate i.e. 342 system clocks to match the audio input sampling rate.

7.4.3 testSPIspeed

This function tests the size of the SRAM buffer and the QSPI transfer rate. If the data in the buffer is out-with maximum or minimum limits, then the QSPI transfer rate must be adjusted to allow the buffer to return to mid-position. Since the QSPI transfer rate cannot be adjusted until transfer is complete, this routine merely disables QSPI wrap-around mode and sets a parameter to indicate the required speed to the main polling routine.

7.4.4 aout_MB6_ISR

This routine receives eight audio data bytes from the CAN and places them at the end of the SRAM buffer. All five audio IDs as transmitted by the five message buffers are filtered and captured by this single message buffer at the output node. The data will be pulled from the opposite end of the buffer and transferred to the QSPI by the main routine.

7.4.5 Audio-out main routine

Here the QSPI is updated as described in Section 5.2.6 and Section 5.2.7. Data is transferred from the SRAM buffer in blocks of eight bytes to either the top or bottom half of the 16-word QSPI queue to ensure that the output transfer mechanism is continuous. The main loop polls the QSPI's CPTQP pointer and SPIF flag to verify when transfer of the upper or lower half of the queue is complete. In addition, the main routine is responsible for adjusting QSPI transfer rate as directed by 'testSPIspeed' to ensure synchronization with the audio input node. A flowchart of this routine is shown in Figure 16.

7.5 Source Listings

7.5.1 ain.c

```
/*****************************************************************************************/
/*                                                                  COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         ain.c                                                          */
/*                                                                                         */
/*                          Audio input function for 68376: QADC ->TouCAN                  */
/*                                                                                         */
/*       INCLUDE FILES:     regs.h    (68376 registers plus basic type defs)               */
/*                          toucan1.h (Constants and bit defs for TouCAN)                  */
/*                          audio.h   (General constants for the audio demo)               */
/*                                                                                         */
/*=========================================================================================*/
/*                                                                                         */
/*       DESCRIPTION:                                                                      */
/*       This file performs the audio in functionality on 68376.                           */
/*       Stereo audio data is sampled on two QADC pins (PQA0,1) into a single              */
/*       40-word queue. Only the upper 8-bits of the 10-bit QADC conversion are            */
/*       used due to the limited CAN bandwidth of approx. 550kbit/s using a                */
/*       1Mbit/s bit rate. On completion of the QADC conversion queue, the 40              */
/*       bytes are transferred into five TouCAN message buffers for transmission           */
/*       over the CAN bus. The main loop is a 'do-nothing' loop as both the                */
/*       QADC and TouCAN are interrupt driven.                                             */
/*                                                                                         */
/*       This code is intended for demonstration purposes only and is not                  */
/*       guaranteed to function in a given application.                                    */
/*                                                                                         */
/*=========================================================================================*/
/*                                                                                         */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                         */
/*                                                                                         */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                            */
/*       LOCATION: East Kilbride, Scotland.                                                */
/*                                                                                         */
/*       UPDATE HISTORY                                                                    */
/*       REV      AUTHOR    DATE      DESCRIPTION OF CHANGE                                */
/*       ---      --------- -------   ---------------------                                */
/*       1.0      A. Dobbin 10/Jun/98 First release of file.                               */
/*                                                                                         */
/*****************************************************************************************/

/* include files */
        #include "regs.h"           /* 68376 registers plus basic type defs*/
        #include "toucan1.h"        /* Constants and bit defs for TouCAN*/
        #include "audio.h"          /* General constants for the audio demo*/

/*****************************************************************************************/
/* Global variables */
/*****************************************************************************************/
volatile char can_busy=0;
volatile char ram_full=0;
volatile union BYTE4 buffer[10];     /* ain buffer - access as 40 bytes or 10 longs*/
```

```
/*****************************************************************************************************/
/* Function prototypes */
/*****************************************************************************************************/
void ain_toucan_init(void);          /* initialise TouCAN MB0-4 for Tx, interrupt on MB4 */
void ain_QADC_init(void);            /* init QADC, 40 word queue, interrupt on Q2 full */

/* ISRs */
void ain_Q2_ISR(void);               /* ISR for QADC Q2 full - AIN */
void ain_MB4_ISR(void);              /* ISR for CAN MB4 Tx completed - AIN */

/*****************************************************************************************************/
/* Main routine */
/*****************************************************************************************************/

void main (void)
{

        /* Initialise TouCAN and QADC modules */

        ain_toucan_init();           /* initialise TouCAN MB0-4 for Tx, interrupt on MB4 */
        ain_QADC_init();             /* init QADC, 40 word queue, interrupt on Q2 full */
        set_ip(3);                   /* enable interrupts of level 4 and above */

        /* Loop forever - audio input routine is run by interrupt service routines */

        for(;;)
        ;

}       /* End of MAIN function */

/*****************************************************************************************************/
/* Function declarations */
/*****************************************************************************************************/

/*=====================================================================================================*/
/*****************************************************************************************************
Function         : ain_toucan_init()
Parameters       : Void
Return           : Void
Description      : TouCAN module initialisation routine for AUDIO IN (QADC ->
                TouCAN). Five message buffers are initialised for audio transmit.
                MB4 will be the last buffer to transmit and it has its interrupt
                enabled for tx completion.
*****************************************************************************************************/

void ain_toucan_init(void)
{
        WORD *buf;

        toucan_init_global ();/* Initialise TOUCAN global registers, disable all MBs */
        IMASK = BIT4;                /* enable buffer 4 interrupts */

        MB0.id = AUDIO0_ID;          /* MB0 has Audio0 id */
        MB1.id = AUDIO1_ID;          /* MB1 has Audio1 id */
        MB2.id = AUDIO2_ID;          /* MB2 has Audio2 id */
        MB3.id = AUDIO3_ID;          /* MB3 has Audio3 id */
        MB4.id = AUDIO4_ID;          /* MB4 has Audio4 id */
```

```
        buf = &MB0.control;
        while (buf <= &MB4.control)  /* scan through MB0-MB4 control words */
        {
                *buf = TX_ENABLE + 8; /* set each MB for Tx with 8 data bytes */
                buf = buf + 8;       /* point to next MB */
        }
        toucan_bus_on ();            /* turn on TouCAN - release all MB's and clear HALT */
}

/*=============================================================================================== */
/*************************************************************************************************
Function         : ain_QADC_init()
Parameters       : Void
Return           : Void
Description      : QADC initialsation routine for AUDIO IN (QADC -> TouCAN).
************************************************************************************************* /

void ain_QADC_init(void)
{
        WORD *ccw;

        QADCMCR = 0x008F;            /* interrupt arbitration level 15 (high) */
        QADCINT = 0x0554;            /* Queue 2 uses IRQ 5 (same as TouCAN), interrupt vector */
                                     /* 84 i.e. offset = $150 (q2 comp = $154) */
        QACR0 = 0x00C5;              /* no mux, PSH=13, PSL=6 i.e. QCLK = FSYS/19. Fsys = 20MHz */
                                     /* => Sample rate = 58.5KHz i.e. 342 clocks (18 clock conversion)
*/
        QACR1 = 0x0000;              /* Queue 1 disabled */

        for (ccw=CCW; ccw<(CCW+40); ccw+=2) /* for each pair of control words */
                {
        *ccw   = 0x0034;             /* odd sample's on PQA0 (audio left) */
        *(ccw+1) = 0x0035;           /* even sample's on PQA1 (audio right) */
                }

        QACR2 = 0xB100;              /* Enable Q2 comp. interrupts, start cont. scan, 40 word queue */
}

/*************************************************************************************************** */
/* ISR Function declarations */
/*************************************************************************************************** */

/*=============================================================================================== */
/*************************************************************************************************
Function         : ain_Q2_ISR()      INTERRUPT SERVICE ROUTINE
Parameters       : Void
Return           : Void
Description      : QADC queue 2 full interrupt service routine for AUDIO IN.
                If TouCAN is busy then the 40 results are stored in the
                RAM buffer. Otherwise they are transferred to the TouCAN.
                A loop is NOT used here so that the interrupt routine is as fast as possible.
************************************************************************************************* /

#pragma interrupt()                  /* declare that next function is an interrupt function */
void ain_Q2_ISR(void)
{

        if (!can_busy)
```

```
{                /*************** if TouCAN is not busy *****************/
                 /* move all 40 Conversions from QADC          */
                 /* Left Justified Signed Result table (high byte only) */
                 /* to TouCAN Message Buffers 0 to 4          */

        MB0.data.b[0] = LJSRR.no[0];                                    /* BUFFER 0 */
        MB0.data.b[1] = LJSRR.no[2];
        MB0.data.b[2] = LJSRR.no[4];
        MB0.data.b[3] = LJSRR.no[6];
        MB0.data.b[4] = LJSRR.no[8];
        MB0.data.b[5] = LJSRR.no[10];
        MB0.data.b[6] = LJSRR.no[12];
        MB0.data.b[7] = LJSRR.no[14];

        MB1.data.b[0] = LJSRR.no[16];                                   /* BUFFER 1 */
        MB1.data.b[1] = LJSRR.no[18];
        MB1.data.b[2] = LJSRR.no[20];
        MB1.data.b[3] = LJSRR.no[22];
        MB1.data.b[4] = LJSRR.no[24];
        MB1.data.b[5] = LJSRR.no[26];
        MB1.data.b[6] = LJSRR.no[28];
        MB1.data.b[7] = LJSRR.no[30];

        MB2.data.b[0] = LJSRR.no[32];                                   /* BUFFER 2 */
        MB2.data.b[1] = LJSRR.no[34];
        MB2.data.b[2] = LJSRR.no[36];
        MB2.data.b[3] = LJSRR.no[38];
        MB2.data.b[4] = LJSRR.no[40];
        MB2.data.b[5] = LJSRR.no[42];
        MB2.data.b[6] = LJSRR.no[44];
        MB2.data.b[7] = LJSRR.no[46];

        MB3.data.b[0] = LJSRR.no[48];/* BUFFER 3 */
        MB3.data.b[1] = LJSRR.no[50];
        MB3.data.b[2] = LJSRR.no[52];
        MB3.data.b[3] = LJSRR.no[54];
        MB3.data.b[4] = LJSRR.no[56];
        MB3.data.b[5] = LJSRR.no[58];
        MB3.data.b[6] = LJSRR.no[60];
        MB3.data.b[7] = LJSRR.no[62];

        MB4.data.b[0] = LJSRR.no[64];                                   /* BUFFER 4 */
        MB4.data.b[1] = LJSRR.no[66];
        MB4.data.b[2] = LJSRR.no[68];
        MB4.data.b[3] = LJSRR.no[70];
        MB4.data.b[4] = LJSRR.no[72];
        MB4.data.b[5] = LJSRR.no[74];
        MB4.data.b[6] = LJSRR.no[76];
        MB4.data.b[7] = LJSRR.no[78];

        MB0.control = TX_ENABLE + 8;                    /* activate each message buffer */
        MB1.control = TX_ENABLE + 8;
        MB2.control = TX_ENABLE + 8;
        MB3.control = TX_ENABLE + 8;
        MB4.control = TX_ENABLE + 8;

        can_busy = 1;                          /* then set flag to show that can is busy */
}
```

```
        else
        {         /****** otherwise TouCAN is busy ***********/

                buffer[0].b[0] = LJSRR.no[0];        /* Transfer a justified, signed byte to SRAM */
                buffer[0].b[1] = LJSRR.no[2];    /* The RAM is written a BYTE at a time since the */
                buffer[0].b[2] = LJSRR.no[4];            /* QADC table has values every 2nd byte */
                buffer[0].b[3] = LJSRR.no[6];
                buffer[1].b[0] = LJSRR.no[8];
                buffer[1].b[1] = LJSRR.no[10];
                buffer[1].b[2] = LJSRR.no[12];
                buffer[1].b[3] = LJSRR.no[14];
                buffer[2].b[0] = LJSRR.no[16];
                buffer[2].b[1] = LJSRR.no[18];
                buffer[2].b[2] = LJSRR.no[20];
                buffer[2].b[3] = LJSRR.no[22];
                buffer[3].b[0] = LJSRR.no[24];
                buffer[3].b[1] = LJSRR.no[26];
                buffer[3].b[2] = LJSRR.no[28];
                buffer[3].b[3] = LJSRR.no[30];
                buffer[4].b[0] = LJSRR.no[32];
                buffer[4].b[1] = LJSRR.no[34];
                buffer[4].b[2] = LJSRR.no[36];
                buffer[4].b[3] = LJSRR.no[38];
                buffer[5].b[0] = LJSRR.no[40];
                buffer[5].b[1] = LJSRR.no[42];
                buffer[5].b[2] = LJSRR.no[44];
                buffer[5].b[3] = LJSRR.no[46];
                buffer[6].b[0] = LJSRR.no[48];
                buffer[6].b[1] = LJSRR.no[50];
                buffer[6].b[2] = LJSRR.no[52];
                buffer[6].b[3] = LJSRR.no[54];
                buffer[7].b[0] = LJSRR.no[56];
                buffer[7].b[1] = LJSRR.no[58];
                buffer[7].b[2] = LJSRR.no[60];
                buffer[7].b[3] = LJSRR.no[62];
                buffer[8].b[0] = LJSRR.no[64];
                buffer[8].b[1] = LJSRR.no[66];
                buffer[8].b[2] = LJSRR.no[68];
                buffer[8].b[3] = LJSRR.no[70];
                buffer[9].b[0] = LJSRR.no[72];
                buffer[9].b[1] = LJSRR.no[74];
                buffer[9].b[2] = LJSRR.no[76];
                buffer[9].b[3] = LJSRR.no[78];

                ram_full = 1;                           /* then set flag to show that ram is full */

        }

        QASR &= ~BIT13;                         /* Clear CF2 bit in QASR to service the interrupt */

}
```

```
/*=============================================================================================== */
/*************************************************************************************************
Function         : ain_MB4_ISR()     INTERRUPT SERVICE ROUTINE
Parameters       : Void
Return           : Void
Description      : TouCAN transmit interrupt service routine for AUDIO IN
                   (QADC->TouCAN).If RAM buffer is full then the 40 data bytes are
                   transferred from RAM to the TouCAN module and transmitted. Otherwise the
                   can_busy flag is cleared and the module waits for the QADC to interrupt.
************************************************************************************************* /

#pragma interrupt()        /* declare that next function is an interrupt function */
void ain_MB4_ISR(void)
{
        if (ram_full)      /******************* IF RAM BUFFER IS FULL THEN ******/
                           /* transfer 40 bytes from RAM to TouCAN MBs */
  {

   /* This method of transferring 40 bytes from the buffer to the CAN */
   /* is used as it is faster to transfer the data as 10 long words */
   /* rather than 40 Bytes. */

                MB0.data.l[0] = buffer[0].l;                        /* fill up message buffer 0 */
                MB0.data.l[1] = buffer[1].l;
                MB1.data.l[0] = buffer[2].l;                        /* fill up message buffer 1 */
                MB1.data.l[1] = buffer[3].l;
                MB2.data.l[0] = buffer[4].l;                        /* fill up message buffer 2 */
                MB2.data.l[1] = buffer[5].l;
                MB3.data.l[0] = buffer[6].l;                        /* fill up message buffer 3 */
                MB3.data.l[1] = buffer[7].l;
                MB4.data.l[0] = buffer[8].l;                        /* fill up message buffer 4 */
                MB4.data.l[1] = buffer[9].l;

                ram_full = 0;
                can_busy = 1;

                MB0.control = TX_ENABLE + 8;                        /* activate each of the MBs */
                MB1.control = TX_ENABLE + 8;
                MB2.control = TX_ENABLE + 8;
                MB3.control = TX_ENABLE + 8;
                MB4.control = TX_ENABLE + 8;

        }
        else                       /*** ELSE (RAM ISN'T FULL) ***/
                can_busy = 0;      /* clear the can busy flag  */
                                   /*****************************/

        IFLAG &= 0xFFE0;           /* Read & write IFLAG to clear it. Clear MB0-4. */

}
```

7.5.2 aout.c

```
/*****************************************************************************************/
/*                                                                  COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         aout.c                                                           */
/*                                                                                           */
/*                          Audio output function for 68376: TouCAN -> QSPI                  */
/*                                                                                           */
/*       INCLUDE FILES:     regs.h    (68376 registers plus basic type defs)                 */
/*                          toucan1.h (Constants and bit defs for TouCAN)                    */
/*                          audio.h   (General constants for the audio demo)                 */
/*                                                                                           */
/*===========================================================================================*/
/*                                                                                           */
/*       DESCRIPTION:                                                                        */
/*       This file performs the audio out functionality on 68376.                            */
/*       Audio data is received from CAN into TouCAN MB6 and is then shifted                 */
/*       out on the QSPI to an external DAC for conversion back to the original              */
/*       stereo audio analogue signals. An internal SRAM is used to buffer                   */
/*       several hundred bytes of data. The amount of data within the buffer is              */
/*       monitored and controlled by adjusting the speed of the QSPI transfer.               */
/*       This buffering and dynamic adjustment of QSPI rate allows for slight                */
/*       differences in the frequency of the oscillators used in the input                   */
/*       and output MCUs.                                                                    */
/*                                                                                           */
/*       This code is intended for demonstration purposes only and is not                    */
/*       guaranteed to function in a given application.                                      */
/*                                                                                           */
/*===========================================================================================*/
/*                                                                                           */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                           */
/*                                                                                           */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                              */
/*       LOCATION: East Kilbride, Scotland.                                                  */
/*                                                                                           */
/*       UPDATE HISTORY                                                                      */
/*       REV      AUTHOR    DATE      DESCRIPTION OF CHANGE                                  */
/*       ---      --------- -------   ---------------------                                  */
/*       1.0      A. Dobbin 10/Jun/98 First release of file.                                 */
/*                                                                                           */
/*****************************************************************************************/


/* include files */
#include “regs.h”                    /* 68376 registers plus basic type defs */
#include “toucan1.h”                  /* Constants and bit defs for TouCAN */
#include “audio.h”                    /* General constants for the audio demo */

/*****************************************************************************************/
/* Global variables */
/*****************************************************************************************/
volatile union BYTE4 aout_buffer[BUFF_SIZE]; /* aout buffer - access as bytes or long words*/
volatile int buffer_cnt=0;  /* Count of data currently held in aout_buffer. Units are longword.*/
```

```
/*************************************************************************************************** */
/* Function prototypes */
/*************************************************************************************************** */
void aout_toucan_init(void);         /* initialise TouCAN MB6 for interrupt driven Rx */
void aout_SPI_init(void);            /* init QSPI for AOUT. */
WORD testSPIspeed (void);            /* aout subroutine - tests speed of SPI and changes if reqd. */

/* ISRs */
void aout_MB6_ISR(void);/* ISR for TouCAN RX - MB6 full. AOUT */

/*************************************************************************************************** */
/* Main routine */
/*************************************************************************************************** */

void main (void)
{
        volatile int SPIptr = 0;     /* Points in aout_buffer for SPI to remove data */
        volatile int SPIchanged = 0; /* Variable for debug use - indicates how often SPI speed changes
*/
        WORD SPIupdate;
        int old_ip;
        extern int ip_level;

        /* Initialise TouCAN and QSPI modules */

        aout_toucan_init();          /* initialise TouCAN MB6 for interrupt driven Rx */
        aout_SPI_init();             /* init QSPI for AOUT. */
        set_ip(3);                   /* enable interrupts of level 4 and above */

        /* Loop forever */

        for(;;)
        {
                while (( SPSR & 0x0F ) < 8)
                ;

/* transfer 8 data bytes from RAM buffer to upper half of QSPI TRAN.RAM */
                TRANRAM.data[0] = aout_buffer[SPIptr].b[0];
                TRANRAM.data[2] = aout_buffer[SPIptr].b[1];
                TRANRAM.data[4] = aout_buffer[SPIptr].b[2];
                TRANRAM.data[6] = aout_buffer[SPIptr++].b[3];
                TRANRAM.data[8] = aout_buffer[SPIptr].b[0];
                TRANRAM.data[10] = aout_buffer[SPIptr].b[1];
                TRANRAM.data[12] = aout_buffer[SPIptr].b[2];
                TRANRAM.data[14] = aout_buffer[SPIptr++].b[3];

                if (SPIptr > BUFF_SIZE-1)
                          SPIptr = 0;

                old_ip = ip_level;  /* Read current value of ip field */
                set_ip(7);          /* disable interrupts while updating variable */
                buffer_cnt -= 2;    /* Decrement the data counter */
                set_ip(old_ip);     /* enable interrupts of level 5 and above */

/* Here we check the size of the RAM buffer to see if the QSPI speed requires adjustment
  to keep input and output nodes synchronised. If the speed does require to be changed,
  the function clears WREN to halt QSPI continuous transfer and sets SPIupdate to non-
  zero. */
```

```
                SPIupdate = testSPIspeed();             /* Run function before SPIF to save time */

                while (!(SPSR & BIT7))                  /* wait until SPI is finished transmitting*/
                ;

                if (SPIupdate)/* Change speed of SPI while it has stopped (queue comp and wrap =0) */
                {

                        SPCR2 |= BIT14;                             /* Turn on SPI wrap-around */
                        SPCR1 = SPIupdate;  /* Set new SPI speed and activate it by setting SPE */
                        SPIchanged++;                                       /* Debug use only */
                }


                SPSR &= ~BIT7;                                                  /* clear SPIF */

/* transfer 8 data bytes from RAM buffer to lower half of QSPI TRAN.RAM */
                TRANRAM.data[16] = aout_buffer[SPIptr].b[0];
                TRANRAM.data[18] = aout_buffer[SPIptr].b[1];
                TRANRAM.data[20] = aout_buffer[SPIptr].b[2];
                TRANRAM.data[22] = aout_buffer[SPIptr++].b[3];
                TRANRAM.data[24] = aout_buffer[SPIptr].b[0];
                TRANRAM.data[26] = aout_buffer[SPIptr].b[1];
                TRANRAM.data[28] = aout_buffer[SPIptr].b[2];
                TRANRAM.data[30] = aout_buffer[SPIptr++].b[3];


                if (SPIptr > BUFF_SIZE-1)
                        SPIptr = 0;

                old_ip = ip_level;                          /* Read current value of ip field */
                set_ip(7);                        /* disable interrupts while updating variable */
                buffer_cnt -= 2;                                /* Decrement the data counter */
                set_ip(old_ip);                       /* enable interrupts of level 5 and above */

        }

}       /* End of MAIN function */

/*************************************************************************************************** */
/* Function declarations */
/*************************************************************************************************** */

/*=========================================================================================================== */
/*************************************************************************************************
Function : aout_toucan_init()
Parameters: Void
Return          : Void
Description: TouCAN module initialisation routine for AUDIO OUT (TouCAN -> QSPI).
        One message buffer is initialised as the audio receive buffer.
************************************************************************************************* /

void aout_toucan_init(void)
{
        toucan_init_global ();      /* Initialise TOUCAN global registers, disable all MBs */
        IMASK |= BIT6;              /* enable buffer 6 interrupts. */

        MB6.id = AUDIO0_ID; /* set id for any audio msg - mask will be set for all audio msgs */
```

```
        MB6.control = RX_ENABLE;     /* buffer is an active receiver. */

        RXGMSK = AUDIO_MASK;         /* Set global mask to receive all audio IDs */

        toucan_bus_on ();            /* turn on TouCAN - release all MB's and clear HALT */
}


/*=============================================================================================== */
/*************************************************************************************************
Function         : aout_SPI_init()
Parameters       : Void
Return           : Void
Description      : QSPI initialisation routine for AUDIO OUT (TOUCAN -> QSPI).
************************************************************************************************* /

void aout_SPI_init(void)
{

 int index;                          /* for scanning command RAM and transmit RAM */

  QSMCR = 0x0080;                    /* interrupt arbitration level 0 */
  QILR = 0x0000;                     /* Disable SCI and QSPI interrupts */

  PQSPAR = 0x7B;                     /* Set all pins of PORTQS to serve QSPI/SCI. */
  PORTQS = 0x0000;                   /* Clear data register */
  DDRQS = 0xFE;                      /* Configure PortQS for QSPI operation */

  SPCR0 = 0x8302;                    /* Master mode, 16 bits per transfer, data..
                                     .. changed on rising edge, SCK = Fsys/4 i.e. 4MHz transfer rate */
  SPCR1 = NORMAL&(~BIT15);           /* Deactivate QSPI and set Tx speed. */
  SPCR2 = 0x4F00;                    /* No interrupts, queue = 0 to 15 with wrap-around.*/
  SPCR3 = 0x00;                      /* No loop or halt mode, no mode fault interrupts.*/

                          /*********** Set up command RAM *********/
        for (index=0; index<16; index+=2)                      /* for each pair of command bytes */
   {
                COMMRAM.data[index]  = 0x7D;            /* first one activates PCS0 (audio left) */
                COMMRAM.data[index+1] = 0x7E;         /* second one activates PCS1 (audio right) */
   }

        for (index=0; index<32; index++)
                TRANRAM.data[index] = 0x00;         * Clear all 32 bytes in the transmit dataRAM */

        SPCR1 |= BIT15;                             /* Activate the QSPI by setting the SPE bit  */
}

/*=============================================================================================== */
/*************************************************************************************************
Function         : testSPIspeed()
Parameters       : Void
Return           : WORD - FAST, NORMAL or SLOW to indicate the reqd. speed for QSPI.
Description      : This routine tests the size of the RAM buffer and indicates
                   whether the speed of the SPI has to alter to compensate. If a speed change
                   is required, QSPI LOOP mode is disabled so that the calling routine can
                   adjust the QSPI rate once transfer stops.
************************************************************************************************* /
```

```
WORD testSPIspeed (void)
{

        if ( (buffer_cnt > BUFF_MAX ) && (SPCR1 != FAST) )
        {
                SPCR2 &= ~BIT14;    /* Disable SPI wrap-around, so SPI will stop at end of queue */
                return(FAST);
        }

        if ( (buffer_cnt > BUFF_MID) && (SPCR1 == SLOW) )
        {
                SPCR2 &= ~BIT14;    /* Disable SPI wrap-around, so SPI will stop at end of queue */
                return(NORMAL);
        }

        if ( (buffer_cnt < BUFF_MID) && (SPCR1 == FAST) )
        {
                SPCR2 &= ~BIT14;    /* Disable SPI wrap-around, so SPI will stop at end of queue */
                return(NORMAL);
        }

        if ( (buffer_cnt < BUFF_MIN) && (SPCR1 != SLOW) )
        {
                SPCR2 &= ~BIT14;    /* Disable SPI wrap-around, so SPI will stop at end of queue */
                return(SLOW);
        }

        return(0);                                  /* Return 0 if RAM buffer data within limits */

}

/*************************************************************************************************** */
/* ISR Function declarations */
/*************************************************************************************************** */

/*=============================================================================================== */
/*************************************************************************************************
Function         : aout_MB6_ISR()    INTERRUPT SERVICE ROUTINE
Parameters       : Void
Return           : Void
Description      : TouCAN receive interrupt service routine for AUDIO OUT.
                Places data from TouCAN MB6 into RAM buffer and updates pointer.
                Interrupt source is serviced and re-enabled for next receive.
************************************************************************************************* /

#pragma interrupt()/* declare that next function is an interrupt function */
void aout_MB6_ISR(void)
{
        WORD tmr_temp;                              /* used to read CAN TIMER and unlock all MBs */
        static volatile int buffer_error = 0;   /* Debug variable - shows overflow of aout_buffer */
        static volatile int CANptr = 0;            /* Points to next free location in aout_buffer */

        tmr_temp = MB6.control;                   /* Read control register to lock receive buffer */

        if (buffer_cnt >= BUFF_SIZE )
        {
                buffer_error++;                /* Debug use - this should never increment past 0 */
        }
```

```
        else
        {
                aout_buffer[CANptr++].l = MB6.data.l[0]; /* move data from message buffer 6 to ram buffer */
                aout_buffer[CANptr++].l = MB6.data.l[1];

                if (CANptr > BUFF_SIZE-1 )
                        CANptr = 0;

                buffer_cnt += 2;                                /* Increment the data counter */
        }

        IFLAG &= ~BIT6;                                         /* Read IFLAG then clear it */

        tmr_temp = TIMER;             /* read the free running timer to release last buffer read */
}
```

7.5.3 toucan.c

```
/*****************************************************************************************/
/*                                                              COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         toucan.c                                                       */
/*                                                                                       */
/*                          Generic functions for TouCAN                                 */
/*                                                                                       */
/* INCLUDE FILES:           regs.h    (68376 registers plus basic type defs)             */
/*                          toucan1.h (Constants and bit defs for TouCAN)                */
/*                          audio.h   (General constants for the audio demo)             */
/*                                                                                       */
/*=======================================================================================*/
/*                                                                                       */
/*       DESCRIPTION:                                                                    */
/*       This file contains generic TouCAN functions as follows :                        */
/*       void toucan_MB_off (void) - Set all 16 MB's to inactive, clear IFLAG            */
/*       void toucan_bus_on (void) - Activates TouCAN by negating the HALT flag          */
/*       void toucan_init_global (void) - Initialise TOUCAN global registers             */
/*       void set_ip (int ip) - Enables interrupts by setting IP in the SR               */
/*                                                                                       */
/*       This code is intended for demonstration purposes only and is not                */
/*       guaranteed to function in a given application.                                  */
/*                                                                                       */
/*=======================================================================================*/
/*                                                                                       */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                       */
/*                                                                                       */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                          */
/*       LOCATION: East Kilbride, Scotland.                                              */
/*                                                                                       */
/*       UPDATE HISTORY                                                                  */
/*       REV      AUTHOR    DATE      DESCRIPTION OF CHANGE                              */
/*       ---      --------- -------   ---------------------                              */
/*       1.0      A. Dobbin 10/Jun/98 First release of file.                             */
/*                                                                                       */
/*****************************************************************************************/
```

```
/* include files */
#include “regs.h”/* 68376 registers plus basic type defs */
#include “toucan1.h”       /* Constants and bit defs for TouCAN */
#include “audio.h”         /* General constants for the audio demo */

/* Global variables */
int ip_level = 7;          /* initial value of ip is 7 after reset */

/*=============================================================================================== */
/*********************************************************************************************
Function         : toucan_MB_off()
Parameters       : Void
Return           : Void
Description      : Set all 16 TouCAN MB's to inactive and clear IFLAG register.
********************************************************************************************* /

void toucan_MB_off (void)
{
        WORD *buf;
        WORD mbiflag;      /* for reading the TouCAN iflag register */

        buf = &MB0.control;
        while (buf <= &MB15.control)                    /* scan through each MB's control word */
        {
                *buf = RX_DISABLE;                         /* and declare each buffer inactive */
                buf = buf + 8;                                      /* point to next buffer */
        }

        mbiflag = IFLAG;
        IFLAG = 0;                  /* Clear all 16 bits in IFLAG register by reading then writing */
}

/*=============================================================================================== */
/*********************************************************************************************
Function         : toucan_bus_on()
Parameters       : Void
Return           : Void
Description      : Unlocks all message buffers and activates the TouCAN by negating
                   the HALT flag to allow synchronisation with the external CAN bus.
********************************************************************************************* /

void toucan_bus_on (void)
{
        WORD tmr_temp;              /* used to read CAN TIMER and unlock all MBs */
        tmr_temp = TIMER;           /* read the free running timer to release last buffer read */
        CANMCR &= ~BIT12;           /* activate the TouCAN module by clearing HALT */
}

/*=============================================================================================== */
/*********************************************************************************************
Function         : toucan_init_global()
Parameters       : Void
Return           : Void
Description      : Initialise TOUCAN global registers (CANMCR, CANICR, CANCTRL0/1,
                   PRESDIV, CANCTRL2) for bit timing, pin control and interrupts. All 16
                   TouCAN MB's are set to inactive. Mask registers and IMASK register are
                   NOT configured here. TouCAN is left in HALT mode.
********************************************************************************************* /
```

```
void toucan_init_global (void)
{

        CANMCR = 0x5400;                                        /* force a SOFT RESET on TouCAN */
        asm("    NOP" );                                      /* Delay a cycle before accessing MCR */
        while ( CANMCR & BIT9 )                                 /* wait for reset cycle to complete */
        ;

        /* Set user def. bus off and error interrupt masks and Rx/Tx pin configuration */
        CANCTRL0 = (BOFFMSK%2 <<7) | (ERRMSK%2 <<6) | (RXMODE%4 << 2) | (TXMODE%4);

        /* Set user def. bits including the prop seg. quanta */
        CANCTRL1 = (SAMP%2 <<7) | (TSYNC%2 <<5) | (LBUF%2 <<4) | (PROPSEG%8);

        /* Configure user defined prescale value */
        PRESDIV = PRES_D%256;

        /* Configure user defined bit timings */
        CANCTRL2 = (RJW%4 <<6) | (PSEG1%8 <<3) | (PSEG2%8);

        /* Enable TouCAN clocks and put into debug mode. Set user defined bits */
        CANMCR = (FRZ%2 <<14) | BIT12 | (WAKEMSK%2 <<10) | (SUPV%2 <<7) | (SELFWAKE%2 <<6) | (APS%2 <<5) | (CAN_IARB%8);

        /* Configure user def. interrupt level and base address */
        CANICR = (ILCAN%8 <<8) | (IVBA%8 <<5);

        toucan_MB_off ();           /* set all 16 TouCAN MB's to inactive */

}

/*=============================================================================================*/
/*********************************************************************************************
Function        : set_ip()
Parameters      : ip - the value to be written to the IP field.
Return          : Void
Description     : This routine enables interrupts of different levels by setting the
                IP field in the status register. Assembler instructions must be used here.
                Note - the value passed is the value written. This will enable interrupts
                of higher levels only i.e. 4 enables IRQ5-7.
*********************************************************************************************/

void set_ip (int ip)
{

        switch (ip) {
                case 0: asm("       ORI.W           #0x0700,SR",
                "         ANDI.W    #0xF0FF,SR" );                          /* Enable level 1-7 */
                                                    break;
                case 1: asm("       ORI.W           #0x0700,SR",
                "         ANDI.W    #0xF1FF,SR" );                          /* Enable level 2-7 */
                                                    break;
                case 2: asm("       ORI.W           #0x0700,SR",
                "         ANDI.W    #0xF2FF,SR" );                          /* Enable level 3-7 */
                                                    break;
                case 3: asm("       ORI.W           #0x0700,SR",
                "         ANDI.W    #0xF3FF,SR" );                          /* Enable level 4-7 */
                                                    break;
```

```
                case 4: asm("       ORI.W               #0x0700,SR",
                "           ANDI.W    #0xF4FF,SR" );                          /* Enable level 5-7 */
                                                        break;
                case 5: asm("       ORI.W               #0x0700,SR",
                "           ANDI.W    #0xF5FF,SR" );                          /* Enable level 6-7 */
                                                        break;
                case 6: asm("       ORI.W               #0x0700,SR",
                "           ANDI.W    #0xF6FF,SR" );                            /* Enable level 7 */
                                                        break;
                case 7: default: asm(" ORI.W            #0x0700,SR",
                "           ANDI.W    #0xF7FF,SR" );                            /* Enable level 7 */
                                                        break;
        }

        ip_level = ip;     /* Record new value of ip */
}
```

7.5.4 audio.h

```
/*****************************************************************************************************/
/*                                                                          COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         audio.h                                                                    */
/*                                                                                                     */
/*                          Header file containing definitions for audio files.                        */
/*                                                                                                     */
/*       INCLUDE FILES:     none                                                                       */
/*                                                                                                     */
/*=====================================================================================================*/
/*                                                                                                     */
/*       DESCRIPTION:                                                                                  */
/*       This is a header file containing definitions for constants used in the                        */
/*       audio input (ain.c) and audio output (aout.c) files.                                          */
/*                                                                                                     */
/*       This code is intended for demonstration purposes only and is not                              */
/*       guaranteed to function in a given application.                                                */
/*                                                                                                     */
/*=====================================================================================================*/
/*                                                                                                     */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                                     */
/*                                                                                                     */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                                        */
/*       LOCATION: East Kilbride, Scotland.                                                            */
/*                                                                                                     */
/*       UPDATE HISTORY                                                                                */
/*       REV      AUTHOR    DATE      DESCRIPTION OF CHANGE                                            */
/*       ---      --------- -------   ---------------------                                            */
/*       1.0      A. Dobbin 10/Jun/98 First release of file.                                           */
/*                                                                                                     */
/*****************************************************************************************************/

/*****************************************************************************************************/
/* Constants for aout */
/*****************************************************************************************************/
#define BUFF_SIZE           200       /* size of aout RAM buffer in long words */
#define BUFF_MAX            180       /* change SPI speed when buffer is 90% full */
#define BUFF_MID            100       /* change SPI speed when buffer is 50% full */
#define BUFF_MIN            20                          /* change SPI speed when buffer is 10% full */
```

```
/* Define QSPI output rate as delay before=22, delay after=8x32=256.
  Total word time = 256 + 22 + (4x16) = 342. This is equal to the QADC
  sampling rate.
  Slow rate = 343 system clocks. Fast rate = 341 system clocks.
  Slow and fast are used to dynamically adjust the QSPI output rate to allow for
  small variations between the two clocks on audio input and audio output.
*/
#define SLOW     0x9708
#define NORMAL   0x9608
#define FAST     0x9508

/*****************************************************************************************************/
/* CAN ID definitions */
/*****************************************************************************************************/
/*       Standard ID format is used i.e. ID's consist of 11 bits.
         Five separate ID's will exist for audio to match the five Tx message
         buffers used in the audio Tx routine (ain).
         The audio mask will 'don't care' the 3-bit audio number (0..4).
*/
#define AUDIO0_ID 0x0000             /* ID=000 0000 0000 RTR=0 RSVD=0000 */
#define AUDIO1_ID 0x0001             /* ID=000 0000 0001 RTR=0 RSVD=0000 */
#define AUDIO2_ID 0x0002             /* ID=000 0000 0010 RTR=0 RSVD=0000 */
#define AUDIO3_ID 0x0003             /* ID=000 0000 0011 RTR=0 RSVD=0000 */
#define AUDIO4_ID 0x0004             /* ID=000 0000 0100 RTR=0 RSVD=0000 */
#define AUDIO_MASK 0xFF0FFFFE        /* Clear MID18-20 to mask the 3 audio lsbs */
```

7.5.5 regs.h

```
/*****************************************************************************************************/
/*                                                                      COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         regs.h                                                                    */
/*                                                                                                    */
/*                          Header file containing definitions for 376 registers.                     */
/*                                                                                                    */
/*       INCLUDE FILES:     none                                                                      */
/*       */
/*====================================================================================================*/
/*                                                                                                    */
/*       DESCRIPTION:*/
/*       This is a header file containing definitions for 376 registers and                           */
/*       type definitions. Only the registers used in the audio functions are                         */
/*       declared here.                                                                               */
/*                                                                                                    */
/*       This code is intended for demonstration purposes only and is not                             */
/*       guaranteed to function in a given application.                                               */
/*                                                                                                    */
/*====================================================================================================*/
/*                                                                                                    */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                                    */
/*                                                                                                    */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                                       */
/*       LOCATION: East Kilbride, Scotland.                                                           */
/*                                                                                                    */
```

```
/*      UPDATE HISTORY                                                                  */
/*      REV     AUTHOR    DATE      DESCRIPTION OF CHANGE                               */
/*      ---     --------- -------   ---------------------                               */
/*      1.0     A. Dobbin 10/Jun/98 First release of file.                              */
/*                                                                                      */
/************************************************************************************** /

/************************************************************************************** */
/* General constants */
/************************************************************************************** */
#define TRUE 0x1
#define FALSE 0x0
#define ENABLE 0x1
#define DISABLE 0x0

/* define register bits */
#define BIT0 0x0001
#define BIT1 0x0002
#define BIT2 0x0004
#define BIT3 0x0008
#define BIT4 0x0010
#define BIT5 0x0020
#define BIT6 0x0040
#define BIT7 0x0080
#define BIT8 0x0100
#define BIT9 0x0200
#define BIT10 0x0400
#define BIT11 0x0800
#define BIT12 0x1000
#define BIT13 0x2000
#define BIT14 0x4000
#define BIT15 0x8000

/************************************************************************************** */
/* Type Definitions */
/************************************************************************************** */
typedef unsigned long LWORD;
typedef volatile unsigned long VLWORD;
typedef unsigned short WORD;
typedef volatile unsigned short VWORD;
typedef unsigned char BYTE;
typedef volatile unsigned char VBYTE;

union BYTE2{                /* Declare a 2 byte union readable as byte, word */
        WORD w;
        BYTE b[2];
};

union BYTE4{                /* Declare a 4 byte union - byte, word or lword */
        LWORD l;
        WORD w[2];
        BYTE b[4];
};

union BYTE8{                /* Declare a 8 byte union - byte, word or lword */
        LWORD l[2];
        WORD w[4];
        BYTE b[8];
};
```

```
/*************************************************************************************************** */
/* Register Definitions */
/*************************************************************************************************** */
/* For 683xx family, 0xFFFFF000 is more efficient than 0x00FFF000.
  Change to 0x007FF000 if MM=1
*/
#define reg_base 0xFFFFF000

/*************************************************************************************************** */
/* QADC Module */
/*************************************************************************************************** */
#define QADC_base (reg_base+0x200)

typedef volatile struct
{
 BYTE no[80];
} byte_table;

typedef volatile struct
{
 WORD no[40];
} word_table;

typedef volatile struct
{
 LWORD no[20];
} lword_table;

#define QADCMCR (*(WORD *) QADC_base)
#define QADCINT (*(WORD *) (QADC_base+0x4))
#define QACR0  (*(WORD *) (QADC_base+0xA))
#define QACR1  (*(WORD *) (QADC_base+0xC))
#define QACR2  (*(WORD *) (QADC_base+0xE))
#define QASR  (*(VWORD *) (QADC_base+0x10))
#define CCW   (WORD *) (QADC_base+0x30)
#define RJURR  (*(byte_table *) (QADC_base+0xB0))
#define RJURRW (*(word_table *) (QADC_base+0xB0))
#define RJURRL (*(lword_table *) (QADC_base+0xB0))
#define LJSRR  (*(byte_table *) (QADC_base+0x130))
#define LJURR  (*(byte_table *) (QADC_base+0x1B0))

#define CF2_CLR 0xDFFF              /* Queue 2 complete flag */

/*************************************************************************************************** */
/* QSM module (QSPI, SCI) */
/*************************************************************************************************** */
#define QSM_base (reg_base+0xC00)

typedef volatile struct             /* allow tran ram to be treated as 32 bytes */
{
 BYTE data[32];
} tr_struct;

typedef volatile struct             /* allow command ram to be treated as 16 bytes */
{
 BYTE data[16];
} cr_struct;
```

```
#define QSMCR  (*(WORD *) (QSM_base+0x00))                              /* General QSM registers */
#define QILR   (*(BYTE *) (QSM_base+0x04))
#define QIVR   (*(BYTE *) (QSM_base+0x05))

#define PORTQS (*(VBYTE *) (QSM_base+0x15))                                     /* port QS */
#define PQSPAR (*(BYTE *) (QSM_base+0x16))
#define DDRQS  (*(BYTE *) (QSM_base+0x17))

#define SPCR0  (*(WORD *) (QSM_base+0x18))                                  /* QSPI sub-module */
#define SPCR1  (*(WORD *) (QSM_base+0x1A))
#define SPCR2  (*(WORD *) (QSM_base+0x1C))
#define SPCR3  (*(BYTE *) (QSM_base+0x1E))
#define SPSR   (*(VBYTE *) (QSM_base+0x1F))
#define TRANRAM (*(tr_struct *) (QSM_base+0x120))
#define COMMRAM (*(cr_struct *) (QSM_base+0x140))

/*************************************************************************************************** */
/* TouCAN module */
/*************************************************************************************************** */
#define TCAN_base (reg_base+0x080)

typedef volatile struct              /* structure to access a can msg buffer */
{
 WORD control;
 WORD id;
 WORD time;
 union BYTE8 data;                   /* allows access to data as byte, word or long */
 WORD reserved;
} can_msg_buf;

#define CANMCR  (*(VWORD *) (TCAN_base+0x00))
#define CANICR  (*(WORD *) (TCAN_base+0x4))
#define CANCTRL0 (*(BYTE *) (TCAN_base+0x6))
#define CANCTRL1 (*(BYTE *) (TCAN_base+0x7))
#define PRESDIV  (*(BYTE *) (TCAN_base+0x8))
#define CANCTRL2 (*(BYTE *) (TCAN_base+0x9))
#define TIMER (*(VWORD *) (TCAN_base+0xA))
#define RXGMSK  (*(LWORD *) (TCAN_base+0x10))
#define RX14MSK  (*(LWORD *) (TCAN_base+0x14))
#define RX15MSK  (*(LWORD *) (TCAN_base+0x18))
#define ESTAT   (*(VWORD *) (TCAN_base+0x20))
#define IMASK (*(WORD *) (TCAN_base+0x22))
#define IFLAG   (*(VWORD *) (TCAN_base+0x24))
#define RXECTR  (*(VBYTE *) (TCAN_base+0x26))
#define TXECTR  (*(VBYTE *) (TCAN_base+0x27))
#define MB0   (*(can_msg_buf *) (TCAN_base+0x80))
#define MB1   (*(can_msg_buf *) (TCAN_base+0x90))
#define MB2   (*(can_msg_buf *) (TCAN_base+0xA0))
#define MB3   (*(can_msg_buf *) (TCAN_base+0xB0))
#define MB4   (*(can_msg_buf *) (TCAN_base+0xC0))
#define MB5   (*(can_msg_buf *) (TCAN_base+0xD0))
#define MB6   (*(can_msg_buf *) (TCAN_base+0xE0))
#define MB7   (*(can_msg_buf *) (TCAN_base+0xF0))
#define MB8   (*(can_msg_buf *) (TCAN_base+0x100))
#define MB9   (*(can_msg_buf *) (TCAN_base+0x110))
#define MB10   (*(can_msg_buf *) (TCAN_base+0x120))
#define MB11   (*(can_msg_buf *) (TCAN_base+0x130))
```

```
#define MB12   (*(can_msg_buf *) (TCAN_base+0x140))
#define MB13   (*(can_msg_buf *) (TCAN_base+0x150))
#define MB14   (*(can_msg_buf *) (TCAN_base+0x160))
#define MB15   (*(can_msg_buf *) (TCAN_base+0x170))
```

7.5.6 toucan1.h

```
/*****************************************************************************************************/
/*                                                                          COPYRIGHT (c) MOTOROLA 1998 */
/*       FILE NAME:         toucan1.h                                                                    */
/*                                                                                                     */
/*       SYNOPSIS:          Header file containing definitions for toucan functions.                     */
/*                                                                                                     */
/*       INCLUDE FILES:     none                                                                         */
/*                                                                                                     */
/*=====================================================================================================*/
/*                                                                                                     */
/*       DESCRIPTION:                                                                                  */
/*       This is a header file containing definitions for constants and bit fields                     */
/*       used in the toucan.c file.                                                                    */
/*                                                                                                     */
/*       This code is intended for demonstration purposes only and is not                              */
/*       guaranteed to function in a given application.                                                */
/*                                                                                                     */
/*=====================================================================================================*/
/*                                                                                                     */
/*       COMPILER: SDS Crosscode      VERSION: 6.7                                                     */
/*                                                                                                     */
/*       AUTHOR: Allan Dobbin         LAST EDIT DATE: 10/Jun/98                                        */
/*       LOCATION: East Kilbride, Scotland.                                                            */
/*                                                                                                     */
/*       UPDATE HISTORY                                                                                */
/*       REV      AUTHOR    DATE      DESCRIPTION OF CHANGE                                            */
/*       ---      --------- -------   ---------------------                                            */
/*       1.0      A. Dobbin 10/Jun/98 First release of file.                                           */
/*                                                                                                     */
/*****************************************************************************************************/

/*****************************************************************************************************/
/* Function prototypes */
/*****************************************************************************************************/
void toucan_MB_off (void);           /* set all 16 TouCAN MB's to inactive */
void toucan_bus_on (void);           /* turn on TouCAN - release all MB's and clear HALT */
void toucan_init_global (void);      /* Initialise TOUCAN global regs and reset all MBs */
void set_ip (int);                   /* Enables interrupts by setting IP in SR */


/*****************************************************************************************************/
/* Constants for TouCAN              */
/*****************************************************************************************************/
/* TouCAN Message Buffer codes */
#define RX_DISABLE          0x00     /* RX buffer not active */
#define RX_ENABLE           0x40     /* RX buffer active */
#define RX_FULL             0x20     /* RX buffer full */
#define RX_OVER             0x60     /* RX buffer overrun */
#define RX_BUSY             0x01     /* RX buffer busy */
```

```
#define TX_DISABLE          0x80      /* TX buffer not active */
#define TX_ENABLE           0xC0      /* TX buffer active */
#define TX_REM_REQ          0xC0      /* TX buffer remote transmission request */
#define TX_REM_REP          0xA0      /* TX buffer remote transmission reply */
#define TX_EN_REM_REP       0xE0      /* TX buffer active, then remote reply */

/*****************************************************************************************/
/*       The following static parameters are passed to the TouCAN
         initialisation function to initialise the global registers */
/*****************************************************************************************/
/* CANMCR register */
/* */
#define FRZ                 DISABLE   /* Enables debug mode during FRZ */
#define WAKEMSK             DISABLE   /* Configure wake-up interrupts */
#define SUPV                0         /* Supervisor access for TouCAN: 0=USER, 1=SUPERVISOR */
#define SELFWAKE            DISABLE   /* Configure self-wake enable */
#define APS                 DISABLE   /* Configure auto power save */
#define CAN_IARB2                     /* TouCAN Interrupt arbitration, 1-15 if interrups enabled */

/* CANICR register */
/* */
#define ILCAN     5                   /* Configure TouCAN IRQ level. 0=disabled, 7=highest priority */
#define IVBA      2                   /* Configure interrupt vector base address, 0-7 */

/* CANCTRL0 register */
/* */
#define BOFFMSK   DISABLE   /* Configure bus off interrupt mask */
#define ERRMSK    DISABLE   /* Configure error interrupt mask */
#define RXMODE    0         /* Configure Rx pin control, 0-3 */
#define TXMODE    0         /* Configure Tx pin control, 0-3 */

/* CANCTRL1 register */
/* Loop mode is no longer supported on TouCAN. */
#define SAMP      1         /* Configure sampling mode: 0=ONE SAMPLE, 1=THREE SAMPLES */
#define TSYNC     DISABLE   /* Configure timer reset on MB0 rx */
#define LBUF      1         /* Configure tx 1st: 0=LOWEST ID, 1=LOWEST MB */
#define PROPSEG   6         /* BIT TIMING: propagation seg (-1): 0-7 */

/* PRESDIV register */
/* */
#define PRES_D    0         /* BIT TIMING: CAN clock prescal divide factor (-1): 0-255 */

/* CANCTRL2 register */
/* */
#define RJW                 3         /* BIT TIMING: resync jump width (-1): 0-3 */
#define PSEG1     5         /* BIT TIMING: phase segment 1 (-1): 0-7 */
#define PSEG2     5         /* BIT TIMING: phase segment 2 (-1): 0-7 */

/* BIT TIMING =

         FSYS                20MHz
-------------------------------------------= ------------- = 1MHz
PRES_D * (1 + PROPSEG(+1) + PSEG1(+1) + PSEG2(+1)) 1 * (1+6+6+7)
*/
```

8 References

The following documents are referred to in this applications note.

1. Motorola CAN document (BCANPSV2.0/D)
2. MC68336/376 User's Manual (MC68336/376UM/AD)
3. The C programming Language, Kernighan & Ritchie, Prentice Hall
4. Programming Microcontrollers in C, Ted Van Sickle, Hightext publications inc.
5. AD1866 specification, Analog Devices
6. PCA82C250 objective specification, Phillips Semiconductors

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor