Application Note

AN1212/D
Rev. 1, 11/2001

J1850 Multiplex Bus Commmunication Using the MC68HC705C8 and the SC371016 J1850 Communications Interface (JCI)

By Chuck Powers
Multiplex Applications

Introduction

The SC371016 J1850 communications interface (JCI) is a serial multiplex communication device developed and manufactured by Freescale for communicating on an automotive serial multiplex bus compatible with the Society of Automotive Engineers Recommended Practice J1850-Class B Data Communication Network Interface. The JCI, which can be easily interfaced to a wide variety of microcontrollers, can be used to transmit and receive serial messages within the framework of J1850, while requiring a minimum of host MCU intervention. The JCI handles all of the communication duties, including complete message buffering, bus access, arbitration and message qualification. Host intervention is only required when a complete message has been received error-free from the multiplex bus, or when the JCI is ready to receive a message for transmission onto the multiplex bus.

This application note describes a basic set of driver routines for communicating on a Class B serial multiplex bus using the JCI and the MC68HC705C8, a multipurpose MCU based upon Motorola's industry standard M68HC05 CPU. Methods will be outlined on interfacing the JCI to the MC68HC705C8, initializing the JCI for proper communication, and transferring data between the JCI and the host MCU. Though these driver routines have been written for use with the MC68HC705C8, the methods described are readily applicable to other microcontroller families.

J1850 Overview

The increase in the complexity and number of electronic components in automobiles has caused a massive increase in the wiring harness requirements for each vehicle. This, in turn, has led to the demand for a means of reducing the amount of wiring needed while at the same time maintaining or improving the communication between various components.

The SAE Recommended Practice J1850 was developed by the Society of Automotive Engineers as a method of medium speed (Class B) serial multiplex communication for use in the automotive environment. Serial multiplex communication (MUX) is a method of reducing wiring requirements while increasing the amount and type of data which can be shared between various components in the automobile. This is done by connecting each component, or node, to a serial bus, consisting of either a single wire or a twisted pair. Each node collects whatever data is useful to itself or other nodes (wheel speed, engine RPM, oil pressure, etc.), and then transmits this data onto the MUX bus, where any other node which needs this data can receive it. This results in a significant improvement in data sharing, while at the same time eliminating the need for redundant sensing systems.

The J1850 protocol encompasses the lowest two layers of the ISO open system interconnect (OSI) model, the data link layer and the physical layer. It is a multi-master system, utilizing the concept of carrier sense multiple access with collision resolution (CSMA/CR), whereby any node can transmit if it has determined the bus to be free. Non-destructive arbitration is performed on a bit-by-bit basis whenever multiple nodes begin to transmit simultaneously. J1850 allows for the use of a single or dual wire bus, two data rates (10.4 kbps or 41.7 kbps), two bit encoding techniques (pulse-width modulation or variable pulse-width modulation), and the use of CRC or Checksum for error detection, depending upon the message format and modulation technique selected.

Features

A J1850 message, or frame, consists of a start of frame (SOF) delimiter, a one- or three-byte header, zero to eight data bytes, a CRC or Checksum byte, an end of data (EOD) delimiter, and an optional in-frame response byte, followed by an end of frame (EOF) delimiter. Frames using a single byte header are transmitted at 10.4 kbps, using VPW modulation, and contain a Checksum byte for error detection (see **Figure 1**). Frames using a one-byte consolidated header or a three-byte consolidated header can be transmitted at either 41.7 kbps or 10.4 kbps, using either PWM or VPW modulation techniques, and contain a CRC byte for error detection (see **Figure 2** and **Figure 3**).

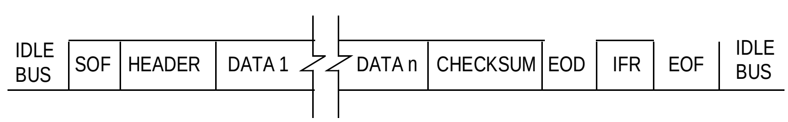

Figure 1. Single Byte Header Frame Format

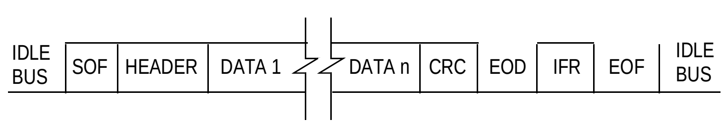

Figure 2. Consolidated One-Byte Header Frame Format

Figure 3. Consolidated Three-Byte Header Frame Format

Each frame can contain up to 12 bytes (PWM) or 101 bit times (VPW), with each byte being transmitted MSB first. The optional in-frame response can contain either a single byte or multiple bytes, with or without a CRC byte.

Table 1 summarizes the allowable features of the J1850 protocol. Which features are used is determined by the requirements of each individual network.

Table 1. J1850 Protocol Options

Feature	1 & 3 Byte Headers	1 & 3 Byte Headers	1 Byte Only Header
Bit encoding	PWM	VPW	VPW
Bus medium	Dual wire	Single wire	Single wire
Data rate	41.7 kbps	10.4 kbps	10.4 kbps
Data integrity	CRC	CRC	Checksum

Frame Headers and Addressing

As outlined above, a J1850 frame can contain one of three types of headers, depending upon a particular system's requirements. The single-byte header incorporates the frame priority/type and target address into a single byte. A one-byte consolidated header also consolidates the frame priority/type and target address into a single byte, with bit 4 = 1 to indicate that it is a one-byte consolidated header. The three-byte header places the frame priority/type into the first byte, the target address of the intended receiver(s) into the second byte, and the source address of the frame originator into the third byte. In the priority/type byte of the three-byte header, bit 4 = 0 to indicate it is a three-byte header.

Frames transmitted on a J1850 network can be either physically or functionally addressed. Since every node on a J1850 network must be assigned a unique physical address, a frame can be addressed directly to any particular node by making the node's physical address the target address of the frame. This is useful in applications such as diagnostic requests, where a specific node's identification may be important. Functional addressing is used when the data being transmitted can be identified by its particular function, rather than its intended receiver(s). With this form of addressing, a frame containing data is transmitted with the function of that data encoded in the target address of the frame. All nodes which require the data of that function can then receive it at the same time. This is of particular importance to networks where the physical address of the intended receivers is not know, or could change, while their function remains the same. An example of data that would be functionally addressed is wheel speed, which could be of interest to multiple receivers, each with a different physical address. Functionally addressing the wheel speed data would allow it to be transmitted to all intended receivers in a single frame, instead of transmitting the data in a separate frame for each receiver.

Error Detection

Every frame transmitted onto a J1850 network contains a single byte for error detection. Frames using the single-byte header contain a Checksum byte, which is the simple summation of all the bytes in the frame, excluding the delimiters and the Checksum byte itself. If the one-byte consolidated header or the three-byte header is used, the frame must contain a cyclical redundancy check, or CRC, byte. This byte is produced by shifting the header and data bytes through a preset series of shift registers. The resulting byte is then inserted in the frame following the data bytes. Any node which receives the frame then shifts the header, data, and CRC bytes through an identical series of shift registers, with an error free frame always producing the result $C4. In most cases, the Checksum calculation and verification will be performed using a software routine, while CRC bytes are generated via hardware. Any frame in which the error detection byte does not produce the proper result is discarded by all receivers, and any in-frame response, if required, is not transmitted.

Arbitration

Arbitration on the multiplex bus is accomplished in a non-destructive manner, allowing the frame with the highest priority to be transmitted, while any transmitters which lose arbitration simply stop transmitting and wait for an idle bus to begin transmitting again. If multiple nodes begin to transmit at the same time, arbitration begins with the first bit following the SOF delimiter, and continues with each bit thereafter. Whenever a transmitting node detects a dominant bit while transmitting a recessive bit, it loses arbitration, and immediately stops transmitting. This is known as "bitwise" arbitration. Since a dominant bit dominates a recessive bit (a "0" dominates a "1"), the frame with the lowest value will have the highest priority, and will always win arbitration, i.e., a frame with priority 000 will win arbitration over a frame with priority 001. This method of arbitration will work regardless of how many bits of priority encoding are contained in the frame. Frequency, messaging strategies are utilized which ensure that all arbitration is resolve by the end of the frame header.

In-Frame Response

The optional in-frame response, or IFR, portion of a frame follows the EOD delimiter, and contains one of three types of information. The first type of IFR contains a single I.D. byte from a single receiver, indicating that at least one node received the frame. The I.D. byte is usually the physical address of the responding node. The second type of IFR contains multiple I.D. bytes from multiple receivers, indicating which receivers actually received the frame.

In this case, the number of response bytes is limited only by the overall frame length constraints. The third type of IFR contains data bytes, with or without a CRC byte, from a single receiver. This type of IFR usually occurs during the IFR portion of a frame in which that data is requested. The CRC byte, if included, is calculated and decoded in an identical manner to the frame CRC, except the transmitter and receiver roles are reversed. In VPW modulation, the in-frame response byte is preceded by a normalization bit, which is required to return the bus to the active state prior to transmitting the first bit of the IFR.

Modulation

As previously mentioned, J1850 frames can be transmitted using two different modulation techniques, pulse width modulation (PWM) or variable pulse width modulation (VPW). The modulation technique used is dependent upon the desired transmission bit rate and the physical makeup of the bus. The PWM technique is primarily used with a bit rate of 41.7 kbps, and a bus consisting of a differential twisted pair. VPW modulation is used with a bit rate of 10.4 kbps and a single-wire bus.

For more detailed information on the features of J1850, refer to *SAE Recommended Practice J1850 – Class B Data Communication Network Interface*. Because this document is still subject to modification, the user should ensure that the most recent revision is referenced.

MC68HC705C8 Microcontroller

The MC68HC705C8 MCU is a multipurpose HCMOS MCU based on the industry standard M68HC05 CPU (refer to **Figure 4**). It contains:

- 8K of erasable programmable read-only memory (EPROM)
- 176 bytes of random-access memory (RAM)
- Serial peripheral interface (SPI) and serial communications interface (SCI) interface ports
- 6-bit timer with one input capture and one output compare,
- On-board computer operating properly (COP) watchdog system

A similar device, the MC68HC05C8, is identical to the MC68HC705C8, except the 8K of EPROM is replaced with 8K of ROM. Some of the major features of the MC68HC705C8 are outlined below. For a detailed description of the features and operation of the MC68HC705C8, refer to the *MC68HC705C8 Technical Data* document.

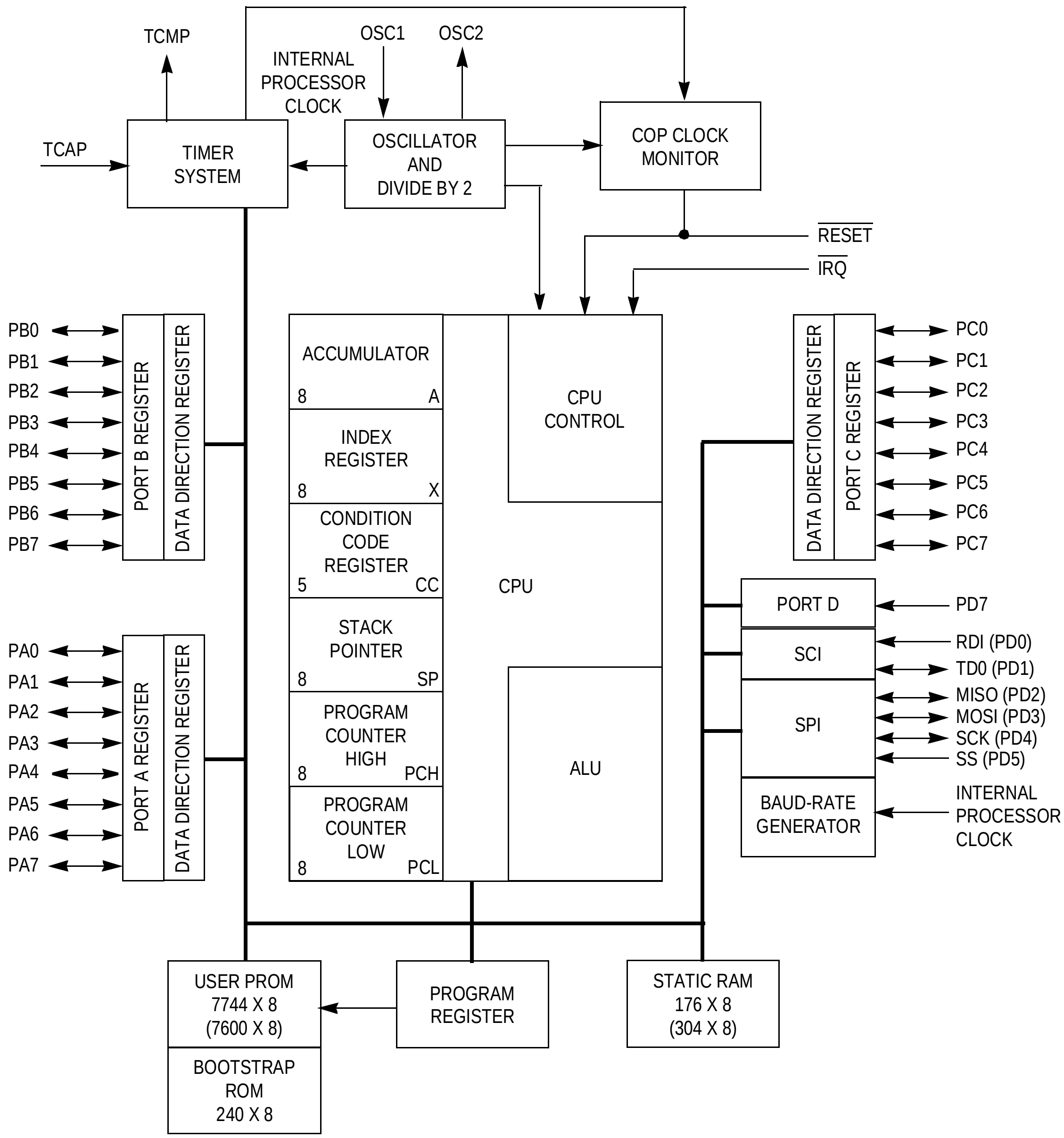

Figure 4. MC68HC705C8 Block Diagram

Memory

The MC68HC705C8 MCU contains 7600 bytes of EPROM (including user-defined reset and interrupt vectors), 223 bytes of bootstrap ROM, and 176 bytes of static RAM. The user can also access up to an additional 144 bytes of user EPROM or 128 bytes of RAM, by programming the RAM1:0 bits of the OPTION register (address $1FDF) on the MC68HC705C8, or by mask option selection on the MC68HC05C8. All ROM and RAM is memory mapped, allowing the user to directly read from (ROM, RAM) or write to (RAM) any memory location. The MC68HC705C8 OPTION register contains a security bit which can be programmed by the user to prevent an unauthorized dump of the contents of the EPROM. **Figure 5** shows the complete memory map of the MC68HC705C8.

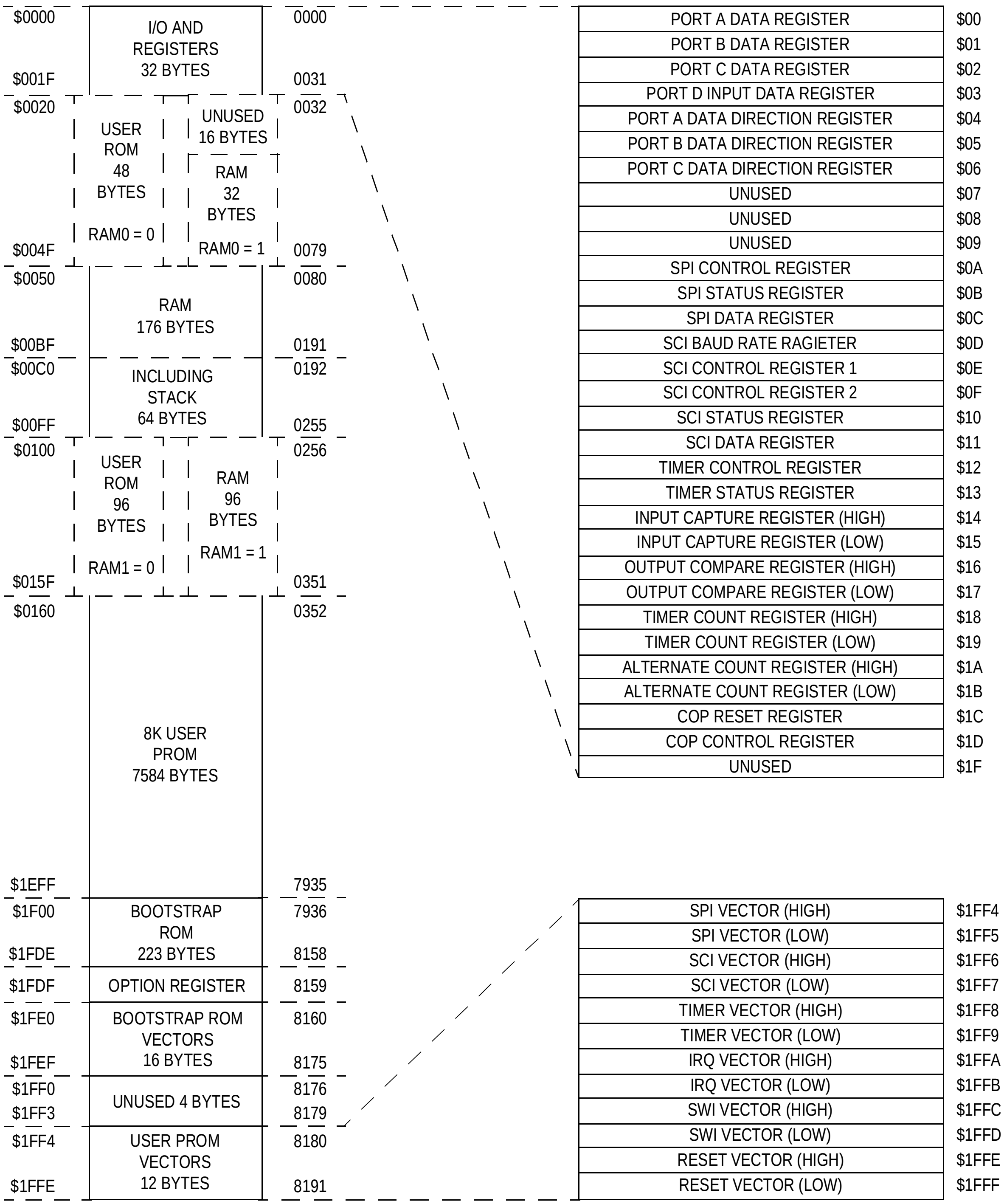

Figure 5. MC68HC705C8 Memory Address Map

Input/Output

The MC68HC705C8 contains 24 bidirectional I/O lines, divided into three 8-bit I/O ports, designated A, B, and C. Port D is a 7-bit input-only port, which shares functions with the serial interface ports. The direction of the 24 I/O lines is controlled by three data direction registers, one for each I/O port. This allows each I/O line to be individually configured by the user as either an input or output. The ports and data direction registers are contained in the first page of the MCU memory map and can be read or written to directly by the user.

Serial Peripheral Interface

The MC68HC705C8 contains a serial peripheral interface (SPI) port, which can be used for high speed serial communication with other peripherals or MCUs. The SPI is a full-duplex, three-wire synchronous serial interface, with programmable clock phase and polarity which can transmit data at up to 1.05 MHz (Master mode).

16-Bit Timer

The MC68HC705C8 contains a timer system featuring a 16-bit free-running counter, one programmable input capture, and one programmable output compare. The timer can be used to record time between input transitions, or to generate output transitions at user specified intervals. The directions for both input edge detection and output edge generation are programmable, and a variety of maskable CPU interrupts are available.

JCI Overview

The SC371016 J1850 communications interface, or JCI, is an all digital device that has been designed to handle all of the necessary communication functions associated with transmitting and receiving frames on a J1850 compatible MUX bus. Through the use of the proper analog transceiver, a single control input, and the correct choice of input oscillator frequencies the JCI can be used to transmit and receive frames in either PWM or VPW modulation, depending upon the user's system requirements. As mentioned above, an external analog transceiver is required to perform the necessary analog waveshaping, output drive, and input compare functions.

When the host MCU has a message ready for transmission onto the MUX bus, the entire message is transmitted to the JCI, and the JCI then performs the necessary bus acquisition, frame transmission, arbitration, and error detection to ensure that only complete, error-free frames are transmitted. When frames are received from the MUX bus, the JCI performs the necessary error checking, determines if the message is of interest to that particular node and, if so, passes the complete message to the host MCU. If desired, the JCI can transmit its physical node address as an in-frame response during the IFR portion of the frame.

The JCI is a CMOS device which can operate over a wide temperature range. It requires a 4 MHz or 8 MHz external oscillator, depending upon the desired transmission bit rate, which can be supplied by a ceramic resonator. **Figure 6** shows a block diagram of the JCI. The following is a description of all major hardware features and functions of the JCI.

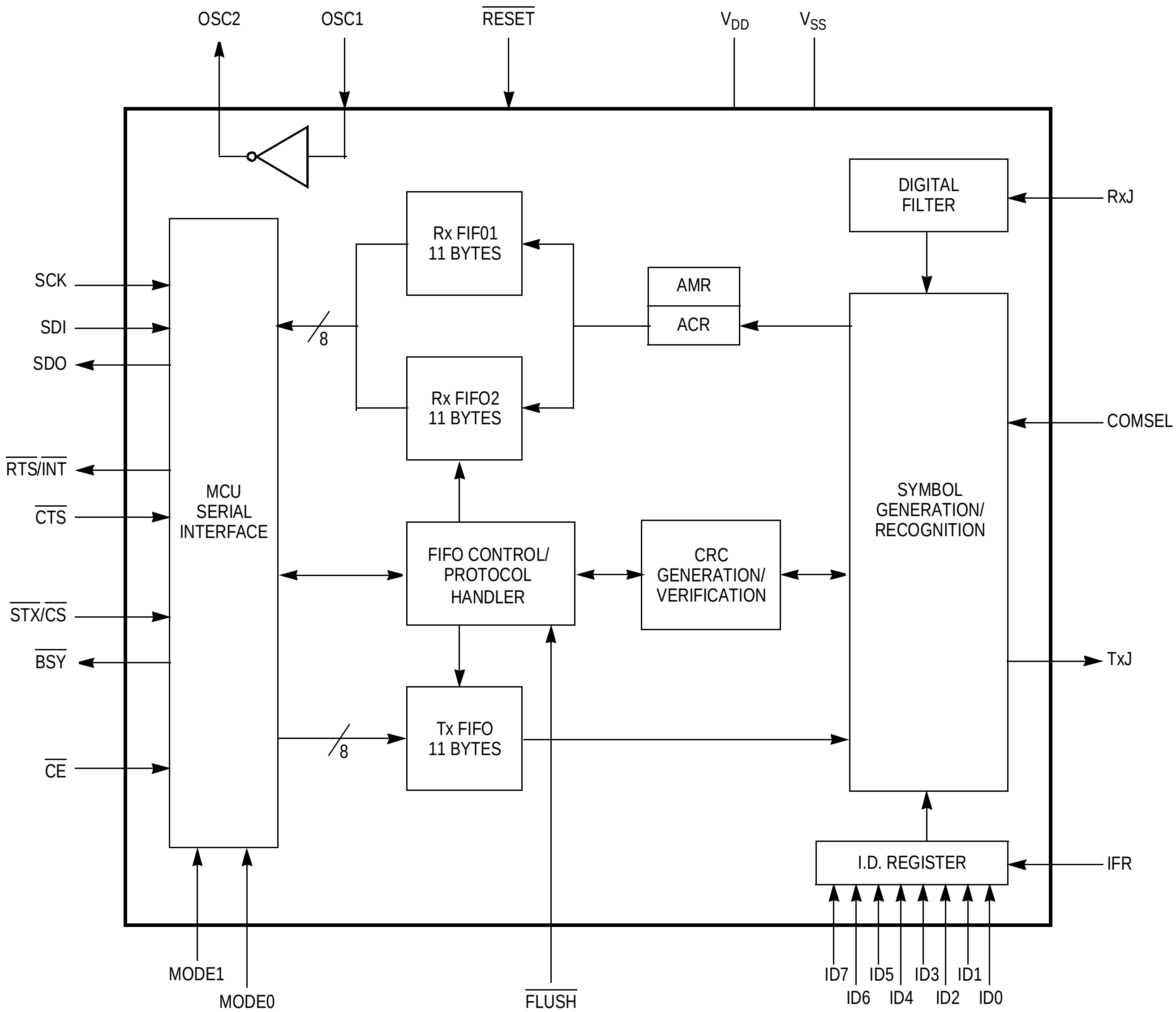

Figure 6. JCI Block Diagram

Host Interface

The JCI has three different serial host interface modes which can be used to interface the JCI to a wide variety of microcontrollers.These three interface modes include:

- Handshake SPI mode
- Handshake SCI mode
- Enhanced SPI mode

These three interface modes provide the host MCU with a choice of two eight-conductor or one five-conductor, high speed, synchronous serial interfaces to the JCI.

The handshake SPI mode is an eight-conductor, full-handshake synchronous serial interface. This mode has three conductors for data transfer, and five for data control and error indication. The three conductor data transfer is compatible with the Freescale serial peripheral interface, preforming an 8-bit "data exchange" between the host MCU and the JCI during each byte transfer. The five control lines are used to delineate data transfers between the host MCU and the JCI, inform the host MCU when the JCI has received a message from the MUX bus, and to indicate to the host when a transmission error has been detected.

The handshake SCI mode is also an eight-conductor, full-handshake synchronous serial interface, having three conductors for data transfer and five for data control and error indication. The three-conductor data transfer is similar in format to the Freescale serial communications interface, although the host MCU must also supply a serial clock to the JCI.

The enhanced SPI mode is a five-conductor synchronous serial interface, compatible with the Freescale serial peripheral interface (SPI). Data is transferred between the host MCU and the JCI in pairs of 8-bit SPI transfers. During the first transfer of each pair, the host MCU transmits a byte of data, which may or may not be valid, to the JCI, while the JCI transmits a status byte to the host MCU, in which is encoded the current status of the JCI. During the second transfer of each pair, the host MCU transmits a command byte to the JCI which can contain a variety of transmit, receive, or general commands, while the JCI transfers a data byte to the host MCU, which may or may not be valid.

For more information on each of these host interface modes, refer to the *J1850 Communications Interface Specification*, Chapter 4: MCU Interface.

JCI Control/ Configuration Inputs

The JCI has 12 inputs that are used to determine:

- The host interface mode
- The message transmission rate
- Modulation technique
- Whether an in-frame response is required
- The physical address of the node

These inputs are normally tied to either a logic 1 or logic 0 in the application, though each can be connected to a host MCU I/O port pin for greater flexibility.

The mode select pins (MODE1:0) are used to determine which interface mode the JCI will use to communicate with the host MCU. Because these pins are level sensitive, the user must take care not to inadvertently change the logic level on one of these inputs, as communication with the JCI will be disrupted. **Table 2** shows the interface mode selection criteria for the mode inputs.

Table 2. JCI Interface Mode Selection

MODE0	MODE1	Operating Mode
V_{SS} V_{SS}	V_{SS} V_{DD}	Enhanced SPI Handshake SPI
V_{DD} V_{DD}	V_{SS} V_{DD}	Handshake SCI Test mode enable

The COMSEL input is used in conjunction with the input oscillator frequency to determine which modulation technique and bit rate the JCI will use to transmit and receive frames on the MUX bus. COMSEL is normally tied to a logic 1 or logic 0 in the application, but it can be connected to a host I/O pin which can be used to control both the logic level of the COMSEL input and an input oscillator control circuit, allowing the user to switch between modulation techniques and transmission bit rates. **Table 3** shows the modulation and bit rate selections as determined by the logic level of the COMSEL input and the input oscillator frequency.

Table 3. Communication Rate and Format Selection

f_{osc}	COMSEL	Communication Baud Rates	Communication Format
8 MHz 8 MHz	V_{DD} V_{SS}	41.7 kbps 20.8 kbps	PWM VPW
4 MHz 4 MHz	V_{DD} V_{SS}	20.8 kbps 10.4 kbps	PWM VPW

The in-frame response (IFR) input determines whether the JCI will transmit, and expect to receive, an in-frame response during the IFR segment of a frame. If the IFR input is at a logic 1, the JCI will transmit its physical node address as a single byte IFR without CRC. The JCI will also expect to receive and IFR each time it transmits a frame onto the MUX bus.

If arbitration is lost while the JCI is attempting to transmit an IFR during the IFR segment of a frame, the JCI will not make another attempt to transmit an IFR within that frame. If the JCI does not receive an IFR during the IFR segment of a message it has transmitted, it will consider that to be a transmission error. If the IFR input is at a logic 0, then the JCI will neither transmit an IFR nor expect to receive an IFR when it transmits a frame onto the MUX bus.

The I.D. inputs (ID7:0) are used to input the physical address of the node. These inputs are normally hardwired in the application to either a logic 0 or logic 1, and are latched into the JCI on the rising edge of a reset pulse.These inputs could be connected to an I/O port of the host MCU, but the JCI would have to be reset by the host MCU each time it wished to change the physical address of the node.

For more information on each of these inputs functions, refer to the *J1850 Communications Interface Specification*, Chapter 3: Operating Modes, and Chapter 4: MCU Interface.

Message Buffers

The JCI contains a single buffer for storing messages for transmission onto the MUX bus, and two buffers for storing messages received from the MUX bus. Each buffer can hold up to 11 bytes, allowing the JCI to transmit and receive the maximum frame length allowed by J1850 (11 bytes + CRC byte).

The transmit (Tx) buffer is an 11-byte buffer into which the host MCU loads all necessary header and data bytes to be transmitted onto the multiplex bus. The CRC byte is calculated and appended onto the frame by the JCI during transmission. The Tx buffer can hold only one complete message at a time. In either handshake interface mode, the host MCU asserts the $\overline{STX}$ input to inform the JCI that new message data is being transmitted and monitors the $\overline{BSY}$ output to determine the status of the Tx buffer. In the enhanced SPI mode, the host MCU loads the Tx buffer through a series of command bytes and monitors the status of the Tx buffer via the status byte.

Once a complete message has been loaded into the Tx buffer, any further attempts by the host MCU to transmit data to the JCI will be ignored until the JCI has transmitted the current frame. Once the data has been emptied from the buffer, the JCI will then accept data for a new message. If the host MCU wishes to transmit a new message to the JCI before the current one has been transmitted, it can empty the Tx buffer by asserting the $\overline{FLUSH}$ input in either handshake interface mode or through use of the "Flush Tx FIFO" command in the enhanced SPI mode.

The receive (Rx) buffers are two11-byte buffers which can each store a complete, maximum length J1850 message (without the CRC). Once the JCI has placed a complete message in an Rx buffer, it makes this Rx buffer available to the host while denying the host access to the other Rx buffer until the next message has been received. Since only one of these Rx buffers can be accessed by the host MCU at a time, to the host there appears to be only a single Rx buffer.

This "ping-pong" action allows the JCI to store a message being received from the MUX bus in one Rx buffer while the host MCU is retrieving a previously received message from the other Rx buffer. Only one message can be stored in each buffer at any one time. In either handshake interface mode, the JCI asserts the $\overline{\text{RTS}}$ output to notify the host MCU that a complete message has been received, and the host MCU asserts the $\overline{\text{CTS}}$ input when it is ready to retrieve each byte. In the enhanced SPI mode, the JCI asserts the $\overline{\text{INT}}$ output when it has received a complete message into an Rx buffer. The host MCU then retrieves the data through a series of command bytes. The host MCU monitors the status of each Rx buffer through the status byte.

Once the JCI has stored a message in each Rx buffer, it will ignore any further frames being transmitted onto the MUX bus until the host MCU has either retrieved the data from, or flushed, one of the Rx buffers. If the host MCU does not wish to retrieve a message from an Rx buffer, it can flush the data, either by using the $\overline{\text{FLUSH}}$ input in either handshake interface mode or with the "Flush Current Rx FIFO" command in the enhanced SPI mode.

Due to the nature of the J1850 bus, each node must receive every frame it transmits to ensure proper arbitration. Therefore, it is possible for the JCI to receive, and pass back to the host, a message it has transmitted. Unless message filtering is used to prevent this, the user’s software must be prepared to deal with this occurrence. However, no in-frame response byte is ever loaded into the Rx buffer or passed back to the host MCU.

For more information on the Rx and Tx buffers, refer to the *J1850 Communications Interface Specification*, Chapter 5: Rx/Tx FIFO’s.

Message Filter

In the enhanced SPI mode, the JCI can utilize a pair of 8-bit registers to filter frames as they are received off of the MUX bus. This allows the JCI to limit the number of messages it receives and thus the amount of host intervention necessary. These registers are called the acceptance code register (ACR) and the acceptance mask register (AMR).

The ACR and AMR are each loaded during initialization, and thereafter, as each frame is being received from the MUX bus, the ACR data is compared to the target address byte of the frame being received. Each bit in the target address byte must match exactly each bit in the ACR for which the corresponding bit in the AMR is set. If the unmasked bits do not match exactly, the remainder of the frame is ignored. Any bits in the target address byte corresponding to bits in the AMR which are not set are not compared. Figure 7 illustrates this procedure.

Message filter is not currently available on the JCI in either handshake interface mode. However, it may be available in the future as a factory mask option.

For more information the JCI's message filtering capabilities, refer to the *J1850 Communications Interface Specification*, Chapter 4: MCU Interface.

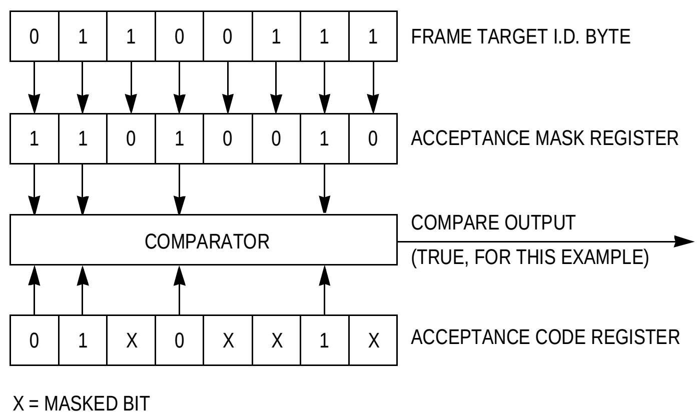

Figure 7. JCI Message Filtering

Error Detection

The JCI uses a variety of methods to ensure the data transmitted onto or received from the MUX bus is error-free. These include a digital input filter, CRC generation and checking, and a constant monitoring of bit and symbol timing, as well as message framing.

All data received from the MUX bus passes through a digital filter. This filter removes short noise pulses from the input signal, which could otherwise corrupt the data being received. The "cleaned up" signal is then passed to the symbol decoder, which decodes the data stream, determining what each bit or symbol is, whether it is of the proper length, and that the message is framed properly.

The CRC byte is calculated by the JCI as it transmits a frame onto the MUX bus and is then appended to the message following the data portion of the frame. The CRC of any frame the JCI receives, including its own, is checked, and if it is not correct, the frame is discarded.

Any frame in which any type of error is detected is discarded by the JCI. If the JCI detects an error while it is transmitting a frame onto the MUX bus, it will immediately halt transmission, wait for an idle bus, and attempt to retransmit the frame. Following the detection of a transmission error, the JCI will attempt to transmit a message up to two more times. Following the third attempt, the

JCI will discard the message, and inform the host MCU that a transmission error has occurred. Loss of arbitration is not considered a transmission error.

For more information on the different methods of error detection and notification used by the JCI, refer to *J1850 Communications Interface Specification*, Chapter 4: MCU Interface and Chapter 7: MUX Interface.

Message Transmitter and Receiver

As mentioned above, the JCI is an all digital device and requires an analog transceiver to supply all transmit waveshaping, transmit drive, and input compare functions. The JCI transmits the frame to the physical interface at digital CMOS levels, where the appropriate waveshaping and drive takes place. Frames being received from the MUX bus are converted back to digital CMOS levels by the analog physical interface and then transmitted to the JCI, where physical interface rise/fall times and propagation delays are taken into account.

For more information on transmitting and receiving messages and transceiver interfacing, refer to *J1850 Communications Interface Specification*, Chapter 7: MUX Interface.

MC68HC705C8/JCI Interface Driver Routines

Communication on the J1850 MUX bus using the JCI can be subdivided into three basic tasks: setup, transmitting, and receiving.

Setup includes:

- Hardware configuration
- Host MCU initialization
- JCI reset
- Loading the ACR and AMR registers with the appropriate message filter data

Transmitting involves:

- Assembling the necessary message bytes
- Transferring the message bytes to the JCI
- Monitoring the JCI to determine when the message has been transmitted successfully

Receiving involves:

- Retrieving message data from the JCI
- Doing any additional filtering
- Storing the data where the user's application software can access it.

These basic driver routines have been divided into these three sections, which should allow the to be more easily understood and used. Each section is detailed below.

This software is intended to be a basic implementation, consuming less than 400 bytes of ROM, so of course the user's system requirements may call for different, and possibly more enhanced, routines. However, these routines should give any potential user a good basic introduction to interfacing the JCI to an MCU, and they can easily be enhanced where added features are needed. Though the MC68HC705C8 is the MCU which was utilized in this example, these driver routines can easily be used with any member of the M68HC05 or M68HC11 Families which has an SPI, a 16-bit timer, and an appropriate amount of memory for the user's application.

This software is written for enhanced SPI mode which requires a little more CPU overhead but fewer host MCU I/O lines. An I/O line of the host MCU is also connected to the RESET input of the JCI, giving the host MCU the ability to reset the JCI through software whenever appropriate. The circuit in **Figure 8** shows a basic JCI/host MCU interface with the JCI configured for 10.4 kbps VPW transmission and IFR required for each message. The physical address of the node depicted is $55. These hardware assumptions are reflected in the software routines, as is the use of three-byte headers, but these routines will work quite well with any hardware configuration required by the user.

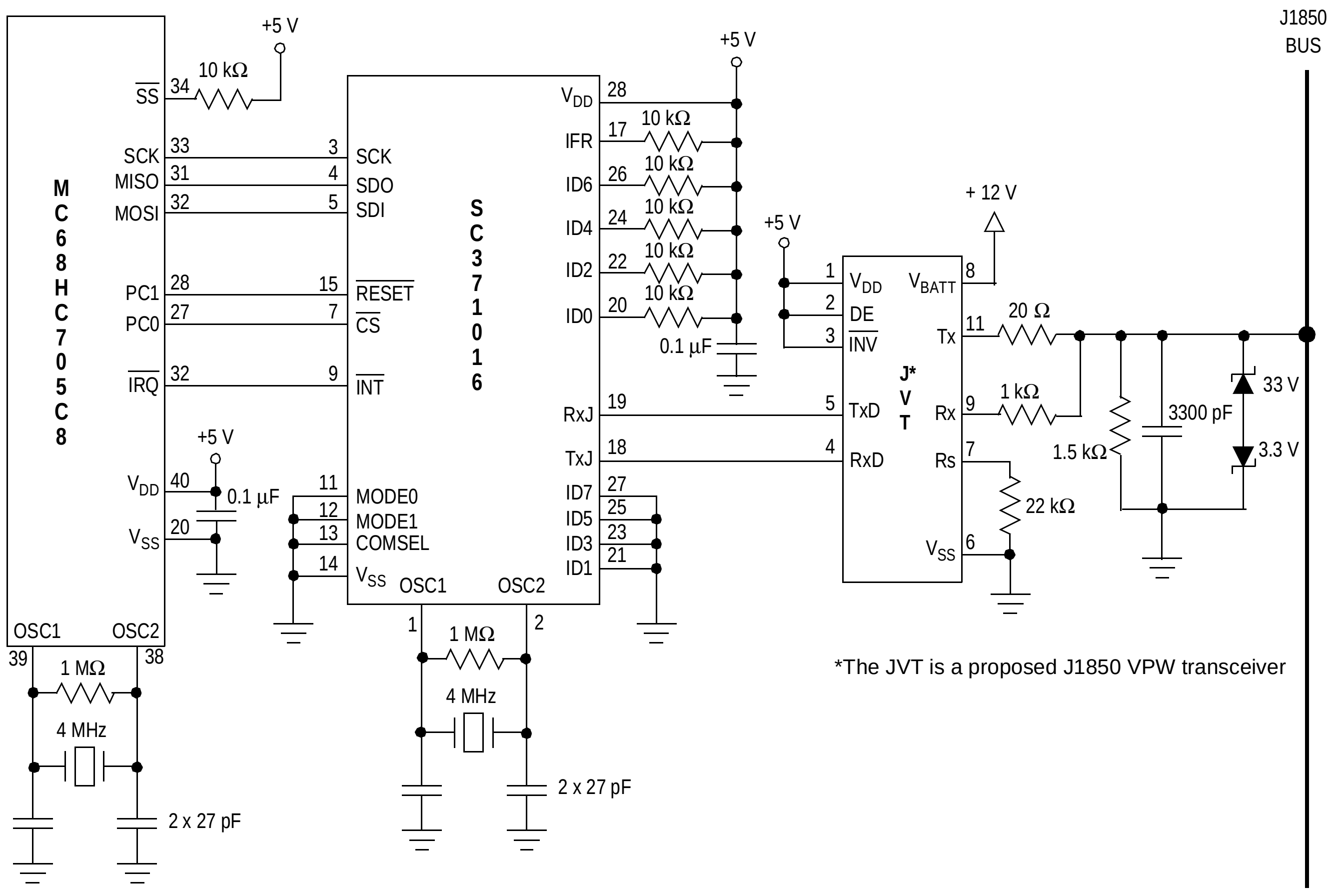

Figure 8. Example MUX Bus Interface Circuit

Setup

This setup routine is in two parts:

- The setup of host MCU RAM
- The reset of the JCI

It should only be necessary to run the setup routines following a host reset, or possibly following the detection of a communication problem on the MUX bus. All of the setup procedures described below can be performed by calling the subroutine JCIRST.

The host MCU RAM is initialized with six bytes reserved for data transfer commands, a single 11-byte transmit message buffer, plus two bytes for transmit control, and a received message buffer pointer and 8-byte received message buffer corresponding to each functional I.D. the user's application must recognize. The use of each RAM location will be explained as it is utilized.

Following a reset of the MC68HC705C8, three host registers are initialized for communication with the JCI. Two port C I/O lines (PC0:1) are configured as outputs, with PC0 connected to the $\overline{CS}$ input of the JCI to control serial communication and PC1 connected to the $\overline{RESET}$ input to allow the host MCU to reset the JCI through software. The serial peripheral interface control register (SPCR) is configured for SPI interrupt disabled, SPI enabled, master mode, CPOL = CPHA = 1, and bit rate configured for 500 kHz SPI communication. The OPTION register is configured for RAM0 = RAM1 = 1 (128 additional bytes of RAM), and the $\overline{IRQ}$ input is programmed for negative edge-sensitivity.

The host MCU must then load the RAM location "txcntrl" with the value $40. "txcntrl" is used for tracking the status of messages transmitted to the JCI and messages transmitted by the JCI onto the MUX bus. The use of "txcntrl" will be explained more fully in **Transmitting**.

The only other initialization required in the host MCU is the initialization of the received message buffer pointers (RMBP). Each RMBP is loaded with the starting address of each corresponding received message buffer. In this example, there are four received message buffers. However, the number of these buffers can be increased, with the only limit being the amount of RAM available and the amount of time the user is willing to spend sorting received messages.

Once the host MCU has completed initializing its internal RAM and registers, the host must perform the necessary initialization of the JCI. This simply involves releasing the $\overline{RESET}$ input, delaying to allow the JCI's internal registers to reset to a known state, and then loading the ACR and AMR registers. The values to be loaded in the ACR and AMR registers are assigned in the equates segment, and each is loaded by calling the subroutines LOADACR and LOADAMR, respectively. Once this is complete and the host MCU clears the I bit, enabling interrupts, the MC68HC705C8 and the JCI are loaded and ready for multiplex communication. Refer to **Figure 9** for a graphical representation of the reset sequence.

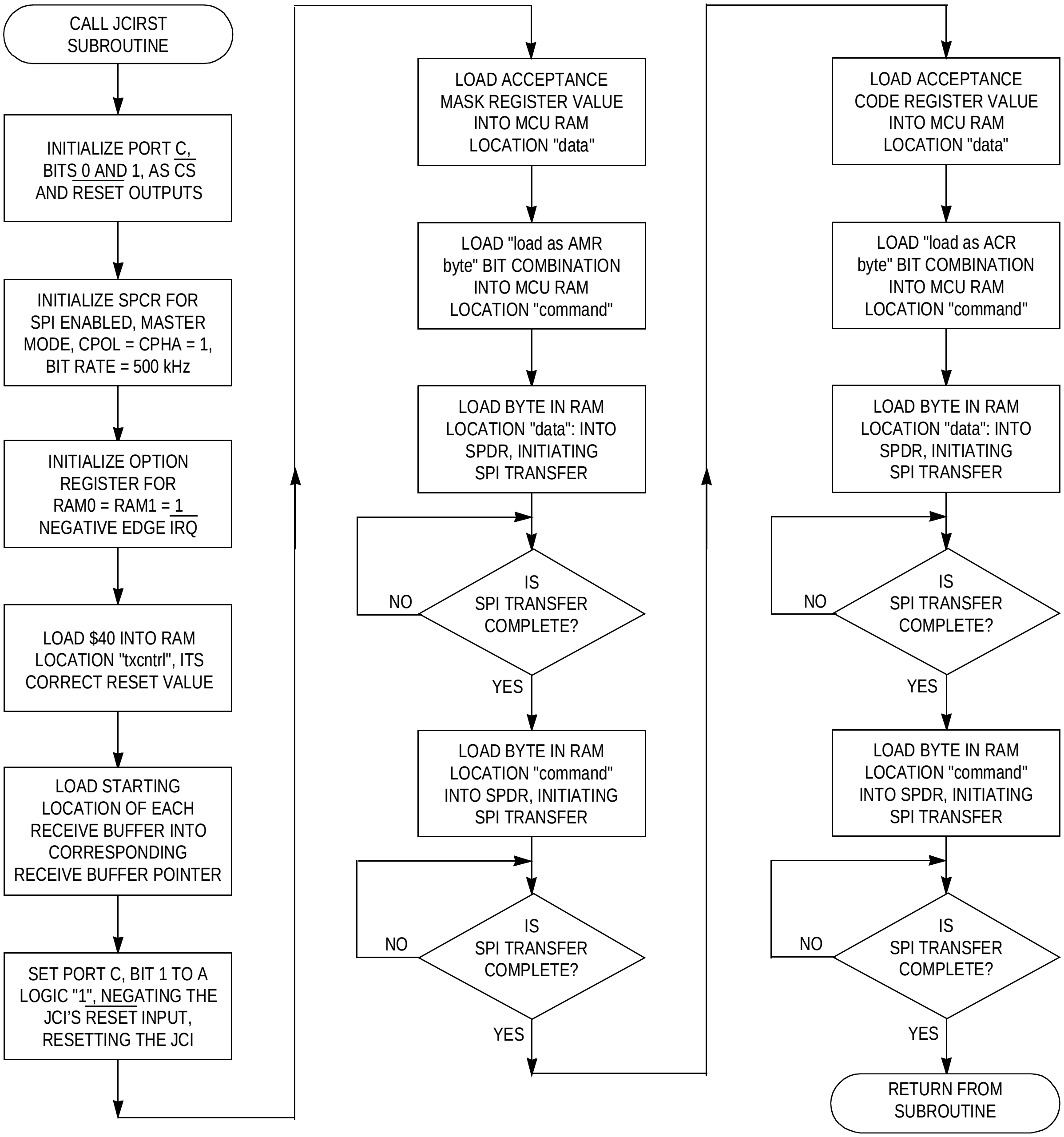

Figure 9. JCIRST Subroutine

Transmitting

Transmitting a message to the JCI for transmission onto the MUX bus simply requires the host MCU to store the message bytes in the correct RAM location and call the TRANSMIT subroutine. The software handles moving the data from the host MCU to the JCI and determining when the message has been transmitted successfully.

When the host MCU has data to be transmitted onto the MUX bus, the 'Message of Tx' bit (labeled "txt") in the RAM location "txcntrl" should first be cleared. This will ensure that a partial message will not inadvertently be transferred to the JCI. The host then stores the message bytes including the header bytes, into RAM, beginning at location "txbuf". The number of bytes in the message is then loaded into the RAM location "txcount". The host then calls the subroutine TRANSMIT. This subroutine will check the status of the JCI to determine whether the previous message has been transmitted and, if so, will transmit the new

message bytes to the JCI for transmission onto the MUX bus and then clear the 'Previous Tx Complete' bit (labeled "txi"). If the previous message has not completed transmission, the TRANSMIT subroutine will set the "txt" bit in the RAM location "txcntrl", and then call a timer subroutine called TIMERSU which enables a timer interrupt to check the JCI status at regular intervals. The TRANSMIT subroutine will then return to the main application routine.

The TIMERSU subroutine reads the current value of the timer's free-running counter, adds a value approximately equal to the shortest valid multiplex frame length, stores the new value in the output compare register, and enables the output compare interrupt. When the counter reaches the output compare value, an interrupt of the CPU occurs. The timer interrupt service routine then checks the status of the JCI. If the previous message has still not completed transmission, the output compare value advance sequence is repeated, and the JCI status is regularly checked, until the current message in the JCI is transmitted onto the MUX bus or is discarded due to reaching the retry limit.

Once the timer interrupt routine determines that the JCI's Tx buffer is empty, the routine checks to see if the "txt" it is set in RAM location "txcntrl". If this bit is set, indicating that a new message is ready for transmission, the "txt" bit is cleared, and the message bytes are transferred to the JCI for transmission, and the timer reset sequence continues.

If the "txt" bit is clear, the timer interrupt routine sets the "txi" bit, and disables the timer interrupt. In this way, bits "txt" and "txi" in RAM location "txcntrl" act as a double semaphore to track the status of both the JCI Tx buffer and the transmit buffer in host MCU RAM, allowing the software to automatically transfer messages to the JCI whenever the Tx buffer in the JCI can accept them.

If the timer interrupt occurs while the host is loading message data into its transmit buffer and the "txt" bit has not been cleared by the user, the number of bytes in RAM location "txcount" will be transferred to the JCI, whether the host has completed updating this data or not. Therefore, the user must ensure that the "txt" bit is cleared before updating data in the host MCU RAM transmit buffer.

Refer to **Figure 10** for a graphical representation of the use of the semaphore bits, and what events cause each bit to be set and cleared.

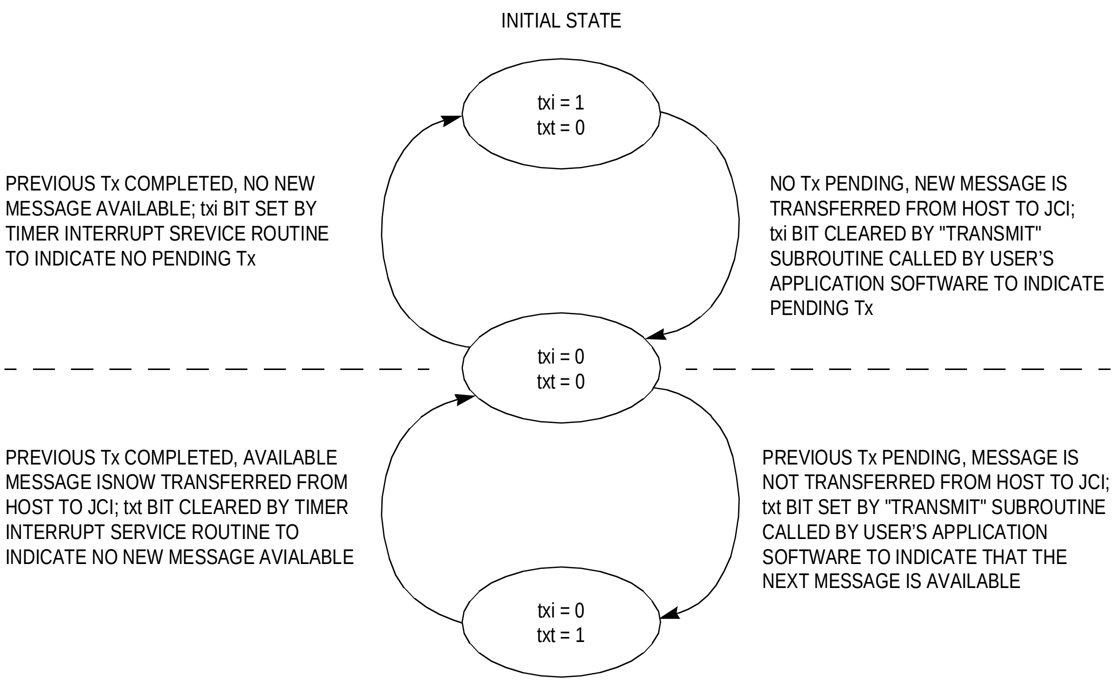

Figure 10. Transmit Double Semaphore State Sequence

If the user's application requires the use of more than one RAM transmit buffer, a transmit queue can easily be set up to transmit messages to the JCI, either in FIFO order, or by priority of the message.

If the host MCU wishes to transmit a message as soon as possible, the Tx buffer in the JCI can be cleared by calling the TXFLUSH subroutine. This subroutine will command the JCI to immediately empty the Tx buffer, preparing it for another message from the host. If the JCI is attempting to transmit when the Tx buffer is flushed, the JCI will abort the transmission, ensuring that the transmission halts on a non-byte boundary.

Figure 11 shows the sequence of the example transmit routine, while **Figure 12** outlines the steps necessary for the actual message transfer between the host MCU and the JCI. **Figure 13** shows the sequence used to check the status of the JCI.

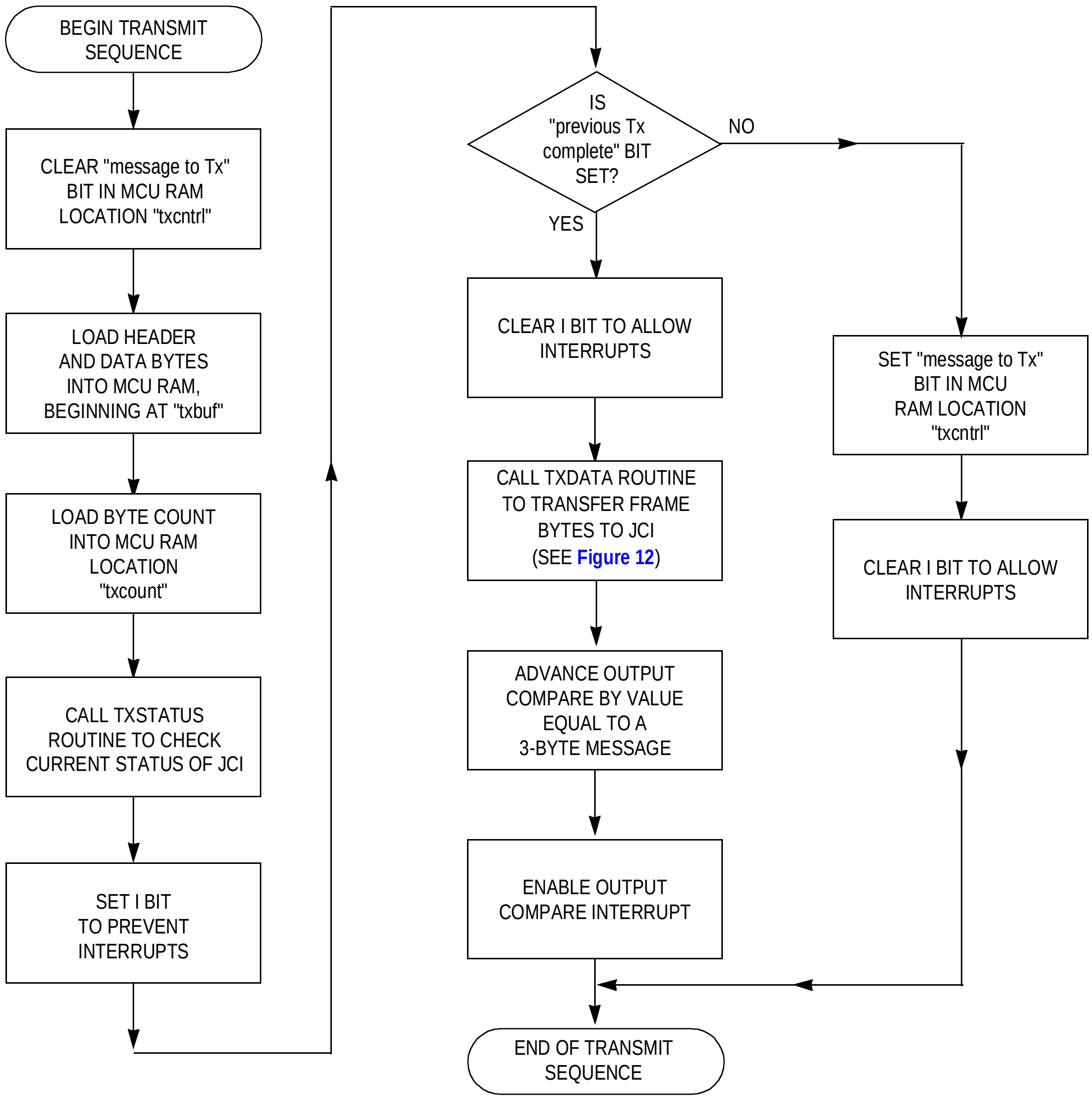

Figure 11. Example Transmit Sequence

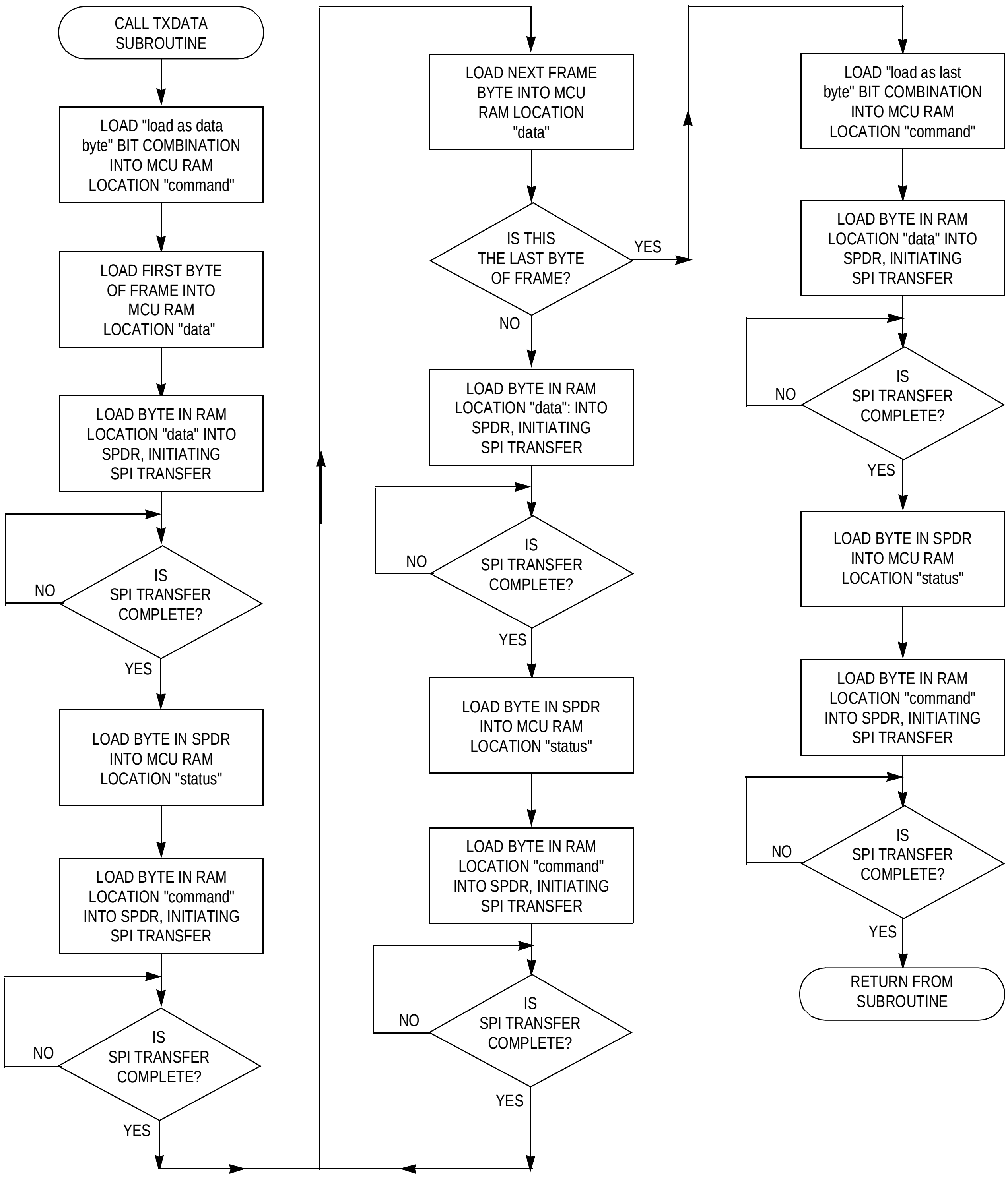

Figure 12. TXDATA Subroutine

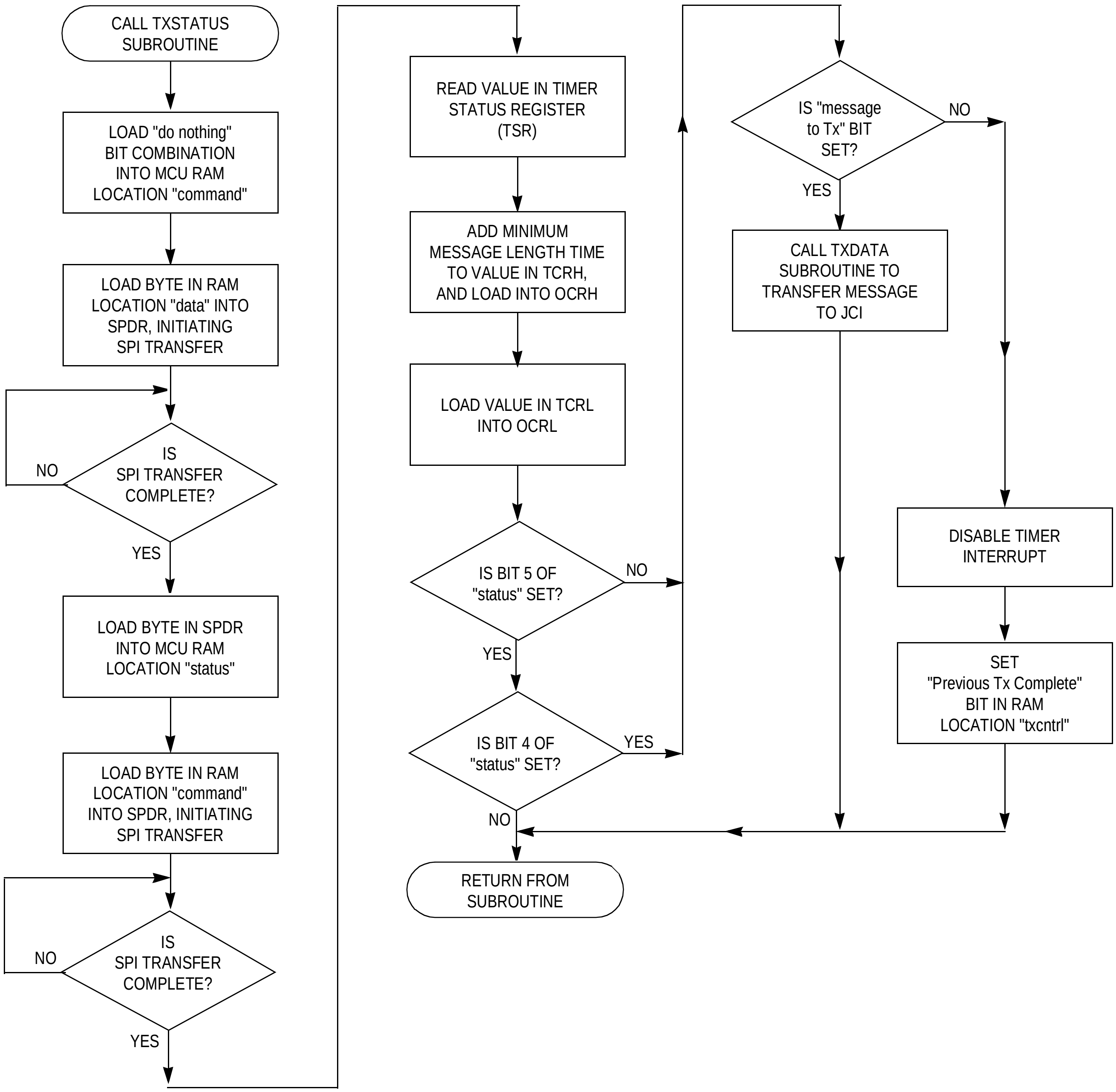

Figure 13. TXSTATUS Subroutine

Receiving

When the JCI has received an error-free message fro the MUX bus which meets the filtering criteria, the $\overline{IRQ}$ interrupt service routine performs the necessary data retrieval, some additional filtering, and then stores the data in a specified location in host RAM where the main application software can access it.

As soon as a qualified message is stored in one of the JCI's two Rx buffers, the INT output is asserted. This output is connected to the MC68HC705C8 IRQ input, generating a CPU interrupt. The interrupt service routine first retrieves and discards the priority/type byte of the message. The second byte of the message, the target address byte, is then retrieved. This byte is compared to each functional I.D. for which a received message buffer has been reserved. As soon as a match is found, the received message buffer pointer corresponding to that functional I.D. is loaded into the X register. The target address byte is then discarded, as is the next byte retrieved, the source address byte. It is not necessary to retain these bytes of the message, since a logical assumption is that the functional I.D. must be known to the receiver already, and the source address is of no use since the function, and not the source, of the message data is what is important.

The data bytes are then retrieved by the host MCU until the JCI status shows the Rx buffer to be empty. Each of the retrieved data bytes is loaded into host MCU RAM beginning at the RAM location whose value is in the appropriate received message buffer pointer. The number of data bytes contained in each received message is not saved because the number of data bytes of any message transmitted on the J1850 MUX bus is pre-defined, and therefore the user should already know how many data bytes will be retrieved for each functional I.D. specified.

At anytime during the retrieval of a message from the JCI, if the host MCU determines that the message is of no interest, the host MCU can call the RXFLUSH subroutine. This subroutine will command the JCI to clear the current Rx buffer immediately. Once the entire message has either been retrieved or cleared from the JCI's Rx buffer, the buffer is released to receive another message from the MUX bus. The interrupt service routine then returns to the point in the user's application software where the interrupt occurred. Refer to **Figure 14** for the sequence followed during the IRQ interrupt service routine.

This procedure results in each host MCU RAM receive buffer containing the latest data received for a specified functional I.D., where the host MCU can access it whenever it needs updated data. Whenever this stored data is accessed, however, the host must first set the I bit to inhibit a receive interrupt. If a receive interrupt is serviced while the host is accessing this stored data, it is possible that the host could end up reading partial data from two different received messages. Also, if physically addressed, or "node-to-node" messages are to be utilized in the user's system, it is a simple matter to modify the receive routine to store the source address of the node-to-node message, if necessary, in the first RAM location of a received message buffer, and to store the number of data bytes received, if necessary, in a temporary storage location for use by the host MCU.

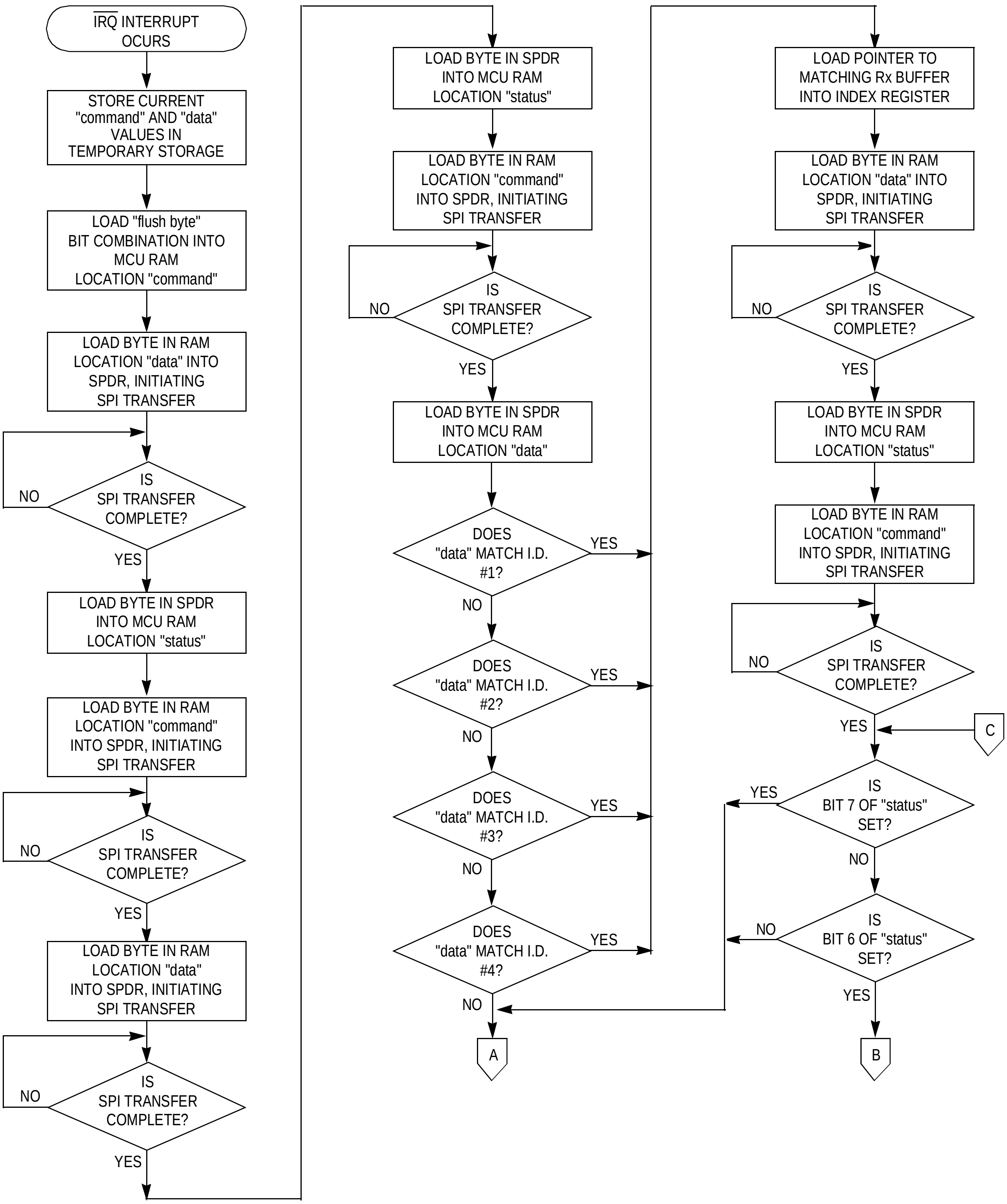

Figure 14. IRQ Interrupt Service Routine (Sheet 1 of 2)

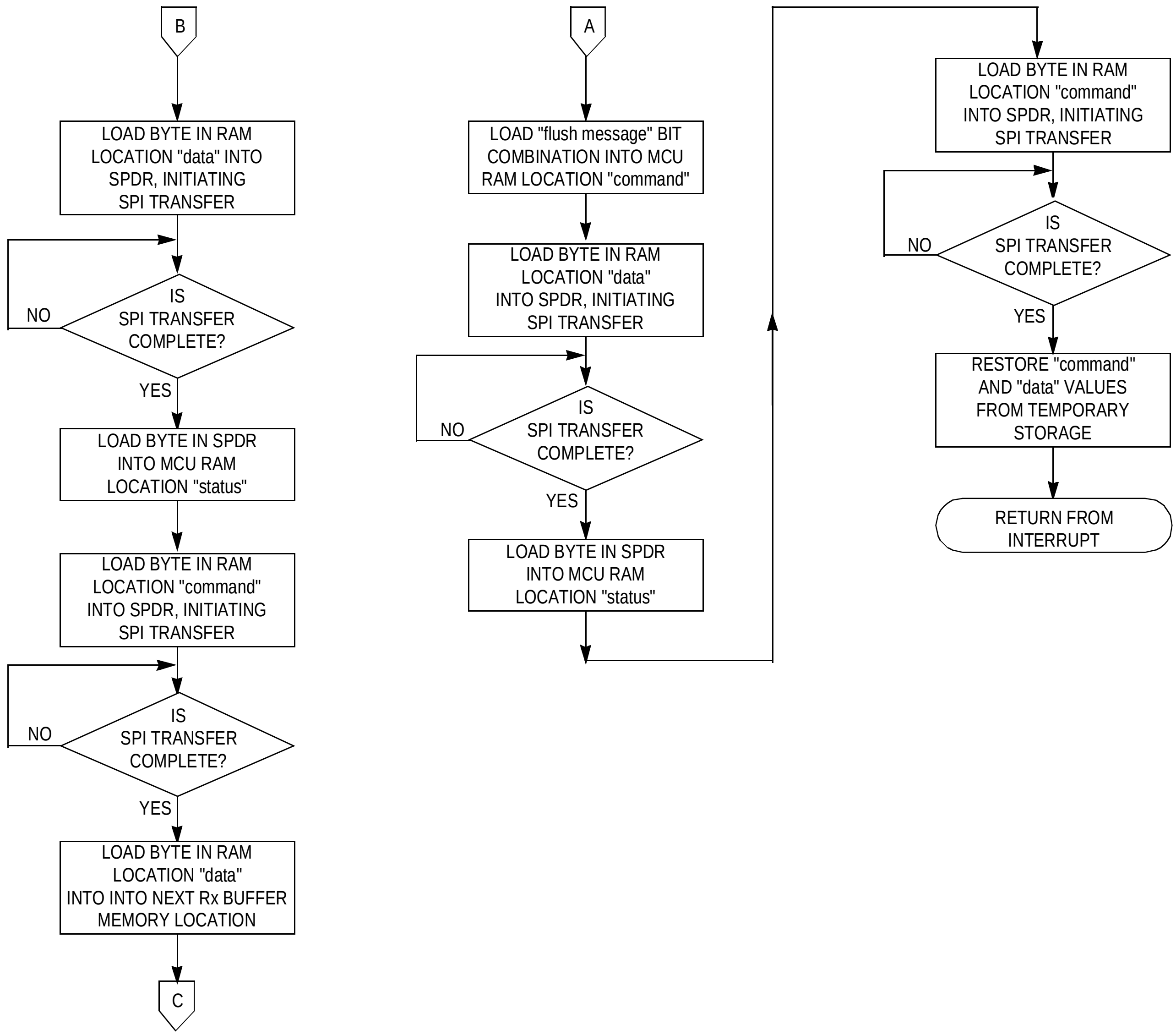

Figure 14. $\overline{\text{IRQ}}$ Interrupt Service Routine (Sheet 2 of 2)

Error Handling

These basic driver routines do not contain extensive error handling procedures. For received messages, the basic assumption made is "if it is no good, don't bother the host with it". Any messages being received which contain errors are simply discarded. Likewise, when transmission or bus errors are detected there are no procedures for dealing with them, since in many instances there is not much the node can do to prevent them from occurring.

However, the JCI can supply the host MCU with extensive transmit, receive, and bus error information, which the host can use to perform any procedures deemed necessary whenever any of these errors are detected on the MUX bus.

Summary

These software driver routines are intended as examples which can be used as a starting point for the development of application software which includes a JCI interface. They should allow the user to quickly construct a basic application using the MC68HC705C8 and JCI for communication onto a J1850 MUX bus, but do not provide a full range of error detection procedures, or otherwise utilize all the information the JCI can provide about the status of the MUX bus, and the messages transmitted and received. For a detailed description of the functions of the JCI, refer to the *J1850 Communications Interface Specification*.

References

J1850 Communications Interface Specification, Revision 1.0, Mot, 1991

M68HC05 Applications Guide, M68HC05AG/AD, Motorola, 1989

MC68HC705C8 Technical Data, MC68HC705C8/D, Motorola, 1990

Society of Automotive Engineers Recommended Practice J1850 – Class B Data Communication Network Interface, J1850, SAE, 1992

Society of Automotive Engineers Recommended Practice J2178 Class B Communication Network Messages, J2178, SAE, 1992

Code Listings

These code listings follow:

- MC68HC705C8/JCI Sample Driver Routines
- MC68HC705C8/JCI Driver Code Example Program

```
          ***********************************************************************
          *                                                                     *
          *                        MC68HC705C8/JCI Sample                       *
          *                           Driver Routines                           *
          *                                                                     *
          *  This code is memory mapped for the MC68HC705C8. It interfaces      *
          *  to the JCI using the Enhanced SPI interface mode.                  *
          *                                                                     *
          *                                                                     *
          *                           Revision History                          *
          *  Rev. 0.1:   (initial release)     Chuck Powers        6/30/92      *
          *  Rev. 0.2:   Add Tx & Rx flush                                      *
          *              routines              Chuck Powers        7/10/92      *
          *  Rev. 0.3:   Fix TXSTATUS and                                       *
          *              message filtering     Chuck Powers        7/17/92      *
          *                                                                     *
          ***********************************************************************

          ***********************************************************************
          *****                          Equates                          *****
          ***********************************************************************

0000      porta     equ  $00       ;Port A
0000      portb     equ  $01       ;Port B
0000      portc     equ  $02       ;Port C
0000      portd     equ  $03       ;Port D

0000      ddra      equ  $04       ;Data Direction, Port A
0000      ddrb      equ  $05       ;Data Direction, Port B
0000      ddrc      equ  $06       ;Data Direction, Port C

0000      spcr      equ  $0a       ;Serial Peripheral Control Register
0000      spsr      equ  $0b       ;Serial Peripheral Status Register
0000      spdr      equ  $0c       ;Serial Peripheral Data Register

0000      tcr       equ  $12       ;Timer Control Register
0000      tsr       equ  $13       ;Timer Status Register
0000      ocrh      equ  $16       ;Timer Output Compare Register (High)
0000      ocrl      equ  $17       ;Timer Output Compare Register (Low)
0000      tcrh      equ  $18       ;Timer Count Register (High)
0000      tcrl      equ  $19       ;Timer Count Register (Low)

          *** TCR Bit Assignments ***

0000      icie      equ  7         ;Input Capture Interrupt Enable
0000      ocie      equ  6         ;Output Compare Interrupt Enable
0000      toie      equ  5         ;Timer Overflow Interrupt Enable

          *** SPCR Bit Assignments ***

0000      spie      equ  7         ;SPCR, Bit 7 - SPI Interrupt Enable
0000      spe       equ  6         ;SPCR, Bit 6 - SPI Enable
0000      mstr      equ  4         ;SPCR, Bit 4 - Master Mode Select
0000      cpol      equ  3         ;SPCR, Bit 3 - Clock Polarity
0000      cpha      equ  2         ;SPCR, Bit 2 - Clock Phase
0000      spr1      equ  1         ;SPCR, Bit 1 - \ SPI Clock
0000      spr0      equ  0         ;SPCR, Bit 0 - / Rate Bits
```

```
              *** SPSR Bit Assignments ***

0000          spif      equ   7          ;SPSR, Bit 7 - SPI Data Transfer Flag
0000          wcol      equ   6          ;SPSR, Bit 6 - Write Collision
0000          modf      equ   4          ;SPSR, Bit 4 - Mode Fault

              *** JCI Control Bit Assignments ***

0000          rst       equ   1          ;Port C, Bit 1 - Reset*
0000          cs        equ   0          ;Port C, Bit 0 - Chip Select*

              *** Port D Bit Assignments ***

0000          ss        equ   5          ;Port D, Bit 5 - Slave Select
0000          sck       equ   4          ;Port D, Bit 4 - Serial Clock
0000          mosi      equ   3          ;Port D, Bit 3 - Master Out, Slave In
0000          miso      equ   2          ;Port D, Bit 2 - Master In, Slave Out

              *** Transmit Control Bit Assignments ***

0000          txt       equ   7          ;Txcntrl, Bit 7 (Message to Tx status)
0000          txi       equ   6          ;Txcntrl, Bit 6 (Previous Tx Complete status)

              *** General Equates ***

0000          ram       equ   $0030      ;Beginning of user RAM
0000          rom       equ   $0180      ;Beginning of user ROM
0000          service   equ   $0300      ;Beginning of Rx IRQ service routine
0000          timer     equ   $0360      ;Beginning of Timer IRQ service routine
0000          vectors   equ   $1ff4      ;Beginning of user vectors
0000          option    equ   $1fdf      ;Option Register Location
0000          none      equ   $0000      ;Bogus Location

              *** JCI Command Byte Equates ***

0000          nothing   equ   $00        ;"Do Nothing" Command Byte
0000          databyte  equ   $04        ;"Load as Data Byte" Command Byte
0000          lastbyte  equ   $0C        ;"Load as Last Byte" Command Byte
0000          maskbyte  equ   $10        ;"Load as I.D. Mask Byte" Command Byte
0000          idbyte    equ   $18        ;"Load as I.D.Byte" Command Byte
0000          flshbyte  equ   $02        ;"Flush First Byte in FIFO" Command Byte
0000          flshfifo  equ   $03        ;"Flush Current FIFO" Command byte
0000          flshtx    equ   $E0        ;"Abort Tx and Flush FIFO" Command byte
0000          tar       equ   $80        ;"Terminate Auto Retry" Command byte

              *** JCI Status Byte Bit Assignments ***

0000          busa      equ   0          ;Bus Status Bit B
0000          busb      equ   1          ;Bus Status Bit A
0000          busact    equ   2          ;Bus Active Bit
0000          tfifoc    equ   3          ;Tx FIFO Status Bit C
0000          tfifob    equ   4          ;Tx FIFO Status Bit B
0000          tfifoa    equ   5          ;Tx FIFO Status Bit A
0000          rfifob    equ   6          ;Rx FIFO Status Bit B
0000          rfifoa    equ   7          ;Rx FIFO Status Bit A
```

```
               *** Timer Interrupt Periods ***

0000           vdelay    equ  $04        ;Counter advance for VPW Tx status routine
0000           pdelay    equ  $02        ;Counter advance for PWM Tx status routine

               *** ACR/AMR Initialization Equates ***

0000           acrbyte   equ  %00100110 ;Init value for ACR
0000           amrbyte   equ  %11010001 ;Init value for AMR

               *** Functional Message I.D.s ***

0000           id1       equ  $00
0000           id2       equ  $20
0000           id3       equ  $04
0000           id4       equ  $24

               **********************************************************************
               *****                 HC05 RAM Storage Assignments                 *****
               **********************************************************************

0030                     org  ram

               *** Data Transfer Storage ***

0030           command   rmb  $1         ;Command Byte Storage
0031           status    rmb  $1         ;Status Byte Storage
0032           data      rmb  $1         ;Data Byte Storage
0033           cmdtemp   rmb  $1         ;Temporary Command Byte Storage
0034           statemp   rmb  $1         ;Temporary Status Byte Storage
0035           datatemp  rmb  $1         ;Temporary Data Byte Storage

               *** Transmit Message Buffer ***

0036           txcount   rmb  $1         ;Host Transmit Message Byte Counter
0037           txbuf     rmb  $b         ;Host Transmit Message Buffer
0042           txcntrl   rmb  $1         ;Host Transmit Message Control Byte

               *** Received Message Buffer Pointer Table ***

0043           msg1      rmb  $1         ;Pointer to RAM holding message w/id1
0044           msg2      rmb  $1         ;Pointer to RAM holding message w/id2
0045           msg3      rmb  $1         ;Pointer to RAM holding message w/id3
0046           msg4      rmb  $1         ;Pointer to RAM holding message w/id4

               *** Received Message Buffers ***

0047           buff1     rmb  $8         ;RAM holding last received message w/id1
004F           buff2     rmb  $8         ;RAM holding last received message w/id2
0057           buff3     rmb  $8         ;RAM holding last received message w/id3
005F           buff4     rmb  $8         ;RAM holding last received message w/id4
```

```
                *************************************************************************
                *                                                                       *
                *                     MC68HC705C8/JCI Driver Code                       *
                *                           Example Program                             *
                *                                                                       *
                * This sample program transmits a message consisting of pri/type=$03,   *
                * target address $73, source address $55, and a data byte, beginning    *
                * with $00. After a delay of 50ms, the data byte is incremented, and    *
                * the message is retransmitted. Anytime a message is received, it       *
                * will be stored in one of the Received Message Buffers, which have     *
                * been reserved for target addresses: $00, $20, $04 & $24 (see          *
                * equates). The Acceptance Mask Register is loaded with $D1, and the    *
                * Acceptance Code Register is loaded with $26. This prevents the        *
                * messages transmitted by the JCI from being received by the JCI,       *
                * and passed back to the host MCU.                                      *
                *                                                                       *
                *************************************************************************

0000            vardata  equ  $70              ;Initialize variable data storage location

0180                     org  rom

0180  CD01AE             jsr  JCIRST           ;Initialize the MC68HC705C8, and reset
                                               ;and initialize the JCI for MUX bus
                                               ;communication

0183  3F70               clr  vardata          ;Clear the location where the variable
                                               ;data byte is stored

0185  1F42      doover:  bclr txt,txcntrl      ;Clear the "Message to Tx" bit to
                                               ;prevent an incomplete message from
                                               ;being transmitted onto the MUX bus

0187  A603               lda  #$03             ;Load the pri/type byte into
0189  B737               sta  txbuf            ;RAM location"txbuf"
018B  A673               lda  #$73             ;Load the target address byte
018D  B738               sta  txbuf+1          ;into RAM location "txbuf"+1
018F  A655               lda  #$55             ;Load the source address byte
0191  B739               sta  txbuf+2          ;into RAM location "txbuf"+2
0193  B670               lda  vardata          ;Load the variable data byte
0195  B73A               sta  txbuf+3          ;into RAM location "txbuf"+3
0197  A604               lda  #04              ;Load the number bytes in the
0199  B736               sta  txcount          ;message into RAM location "txcount"

019B  CD01FE             jsr  TRANSMIT         ;Call the subroutine TRANSMIT,
                                               ;initiating the transmit sequence

019E  9D                 nop

019F  AE3F               ldx  #$3f             ;Delay loop . . .

01A1  A6FF      lp1:     lda  #$ff
01A3  4A        lpo:     deca
01A4  26FD               bne  lpo

01A6  5A                 decx
01A7  26F8               bne  lpl              ;End delay loop.
```

```
01A9   3C70                 inc   vardata         ;Increment the variable data byte
01AB   CC0185               jmp   doover          ;Jump back, and transmit again
                   *************************************************************************
                   *                                                                       *
                   *                             Subroutines                               *
                   *                                                                       *
                   *************************************************************************
                   *************************************************************************
                   *****                  Initialization Subroutine                    *****
                   *************************************************************************
                   *** Initialization of Port C for JCI Handshake ***
01AE   A601        JCIRST:  lda   #%00000001      ;C7-C2 user i/o, C1 - reset*,
01B0   B702                 sta   portc           ;C3-C0 - cs*
01B2   A603                 lda   #%00000011      ;C7-C2 - user assigned
01B4   B706                 sta   ddrc            ;C1=C0 - outputs
                   *** Initialization of SPCR for JCI Serial Comm. ***
01B6   A65D                 lda   #$01011101      ;B7 - spie, B6 - spe, B4 - mstr
01B8   B70A                 sta   spcr            ;B3 - cpol, B2 - cpha, B1:0 - Bit Rate
                   *** Option Reg. IRQ Sensitivity ***
01BA   A6C0                 lda   #%11000000      ;Program RAM0=RAM1=0 for more RAM
01BC   C71FDF               sta   option          ;Program IRQ* for negative edge only
                   *** Clear Txmit Control Register ***
01BF   3F42                 clr   txcntrl         ;This will prepare the transmit control
01C1   1C42                 bset  txi, txcntrl    ;register for Host/JCI communication
                   *** Initialization of Receive Message Buffer Pointers ***
01C3   A647                 lda   #buff1          ;Load location of message buffer w/id1
01C5   B743                 sta   msg1            ;in message buffer pointer msg1
01C7   A64F                 lda   #buff2          ;Load location of message buffer w/id2
01C9   B744                 sta   msg2            ;in message buffer pointer msg2
01CB   A657                 lda   #buff3          ;Load location of message buffer w/id3
01CD   B745                 sta   msg3            ;in message buffer pointer msg3
01CF   A65F                 lda   #buff4          ;Load location of message buffer w/id4
01D1   B746                 sta   msg4            ;in message buffer pointer msg4
                   *** Release JCI Reset* Input ***
01D3   1202                 bset  rst,portc       ;Negate reset
01D5   9D                   nop                   ;Delay to allow
01D6   9D                   nop                   ;All internal registers in
01D7   9D                   nop                   ;JCI to reset
01D8   9D                   nop
```

```
01D9  CD0263            jsr   LOADAMR         ;Call subroutine to load Acceptance Mask
                                              ;Byte into Acceptance Mask Register in JCI
01DC  CD026F            jsr   LOADACR         ;Call subroutine to load Acceptance Code
                                              ;Byte into Acceptance Code Register in JCI
01DF  9A                cli                   ;Clear Host Interrupt Mask Bit
01E0  81                rts                   ;End of JCI init subroutine
             ************************************************************************
             *****                        Other Subroutines                    ******
             ************************************************************************
             *** MC68HC705C8/JCI Data Exchange Subroutine ***
01E1  1102   TRANSFER:  bclr  cs,portc        ;Assert Chip Select*
01E3  B632              lda   data            ;Load data byte in acc.
01E5  B70C              sta   spdr            ;Store in SPI data reg., initiating tx
01E7  3D0B   txwait1:   tst   spsr            ;Is previous transfer complete?
01E9  2AFC              bpl   txwait1         ;loop until done
01EB  B60C              lda   spdr            ;Load received status byte into acc.
01ED  B731              sta   status          ;Store in status byte storage location
01EF  B630              lda   command         ;Load command byte into acc.
01F1  B70C              sta   spdr            ;Store in SPI data reg., initiating tx
01F3  3D0B   txwait2:   tst   spsr            ;Is previous transfer complete?
01F5  1AFC              bpl   txwait2         ;loop until done
01F7  B60C              lda   spdr            ;Load received data byte into acc.
01F9  B732              sta   data            ;Store in Data byte storage location
01FB  1002              bset  cs,portc        ;Negate Chip Select*
01FD  81                rts                   ;Return from subroutine
             *** TRANSMIT Subroutine ***
01FE  CD0233 TRANSMIT:  jsr   TXSTATUS        ;Call TXSTATUS subroutine to check
                                              ;status of previously Tx'ed message
0201  9B                sei                   ;Set I-bit to make sure "PreviousTX
                                              ;Complete" bit is not set before "Message
                                              ;to Tx" bit can be set
0202  0C4206            brset txi,txcntrl,clr6 ;Has Tx completed?
0205  1E42              bset  txt,txcntrl     ;Set txt bit - message to Tx
0207  9A                cli                   ;Clear I-bit
0208  CC0216            jmp   tdone           ;Jump to end of Tx subroutine routine
```

```
020B  9A          clr6:    cli                          ;Clear I-bit

020C  CD0217               jsr   TXDATA                 ;Jump to routine to transmit message
                                                        ;data to JCI

020F  1D42                 bclr  txi,txcntrl            ;Clear txi bit - previous Tx not complt

0211  CD0256               jsr   TIMERSU                ;Call subroutine to setup timer int.

0214  1C12                 bset  ocie,tcr               ;Enable Output Compare Interrupt

0216  81          tdone:   rts                          ;Return from subroutine

                  *** Tx Message Data Transfer Subroutine ***

0217  5F          TXDATA:  clrx                         ;Set X-register to 0

0218  5C          nexttx:  incs                         ;Increment X-register

0219  E636                 lda   txcount,x              ;Load message data byte into
021B  B732                 sta   data                   ;Data storage location

021D  B336                 crz   txcount                ;Compare x-register with # of bytes
021F  270A                 beg   lasttx                 ;If last byte, jump to last byte sequence

0221  A604                 lda   #databyte              ;Load "load as data byte" command
0223  B730                 sta   command                ;into RAM location "command"

0225  CD01E1               jsr   TRANSFER               ;Call TRANSFER subroutine to transfer
                                                        ;data and command bytes to JCI

0228  CC0218               jmp   nexttx                 ;Go get next byte

022B  A60C        lasttx:  lda   #lastbyte              ;Load "load as last byte" command
022D  B730                 sta   command                ;into RAM location "command"

022F  CD01E1               jsr   TRANSFER               ;Call TRANSFER subroutine to transfer
                                                        ;last data and command byte to JCI

0232  81                   rts                          ;Return from subroutine

                  *** Tx Status Check Subroutine ***

0233  A600        TXSTATUS: lda  #nothing               ;Load "do nothing" command
0235  B730                 sta   command                ;into RAM location "command"

0237  CD01E1               jsr   TRANSFER               ;Call TRANSFER subroutine to
                                                        ;retreive current status from JCI

023A  CD0256               jsr   TIMERSU                ;Call TIMERSU subroutine to reset
                                                        ;OC value for timer interrupt

023D  0B3106               brclr tfifoa,status,txdone     ;Is Tx FIFO empty?

0240  083103               brset tfifob,status,txdone     ;Has transmitter made best
                                                          ;attempt to Tx message?
```

```
0243  CC0255            jmp   return          ;Jump to end of subroutine
0246  0F4208   txdone:  brclr tx,txcntrl,set6 ;Message to Tx?
0249  CD0217            jsr   TXDATA          ;Jump to routine to transmit message
                                              ;data to JCI
024C  1F42              bclr  txt,txcntrl     ;Clear txt bit, no message to Tx
024E  CC0255            jmp   return          ;Jump to end of subroutine
0251  1D12     set6:    bclr  ocie,tcr        ;Clear OCIE bit in TCR, disabling int.
0253  1C42              bset  txi,txcntrl     ;Set txi bit, previous Tx complete
0255  81       return:  rts                   ;Return from subroutine
               *** Timer Setup Subroutine ***
0256  B613     TIMERSU: lda   tsr             ;Read TSR
0258  B618              lda   tcrh            ;Load MSB timer value into acc.
025A  AB04              add   #vdelay         ;Add appropriate delay value
025C  B716              sta   ocrh            ;Store in OCR MSB
025E  B619              lda   tcrl            ;Load LSB timer value into acc.
0260  B717              sta   ocrl            ;Store in OCR LSB
0262  81                rts                   ;Return from subroutine
               *** Load Acceptance Mask Register Subroutine ***
0263  A6D1     LOADAMR: lda   #amrbyte        ;Load AMR data byte into
0265  B732              sta   data            ;Data storage location
0267  A610              lda   #maskbyte       ;Load "load as AMR byte" command
0269  B730              sta   command         ;into RAM location "command"
026B  CD01E1            jsr   TRANSFER        ;Call TRANSFER subroutine to transfer
                                              ;data and command bytes to JCI
026E  81                rts                   ;Return From Subroutine
               *** Load Acceptance Code Register Subroutine ***
026F  A626     LOADACR: lda   #acrbyte        ;Load ACR data byte into
0271  B732              sta   data            ;Data storage location
0273  A618              lda   #idbyte         ;Load "load as ACR byte" command
0275  B730              sta   command         ;into RAM location "command"
0277  CD01E1            jsr   TRANSFER        ;Call TRANSFER subroutine to transfer
                                              ;data and command bytes to JCI
027A  81                rts                   ;Return From Subroutine
```

```
                *** Flush Rx FIFO Subroutine ***

027B  A603      RXFLUSH: lda  #flshfifo       ;Load "flush Rx FIFO" command
027D  B730               sta  command         ;into RAM location "command"

027F  CD01E1             jsr  TRANSFER        ;Call "TRANSFER" subroutine to transfer
                                              ;data and command bytes to JCI

0282  81                 rts                  :Return From Subroutine

                *** Flush Tx FIFO Subroutine ***

0283  A6E0      TXFLUSH: lda  #flshtx         ;Load "flush Tx FIFO" command
0285  B730               sta  command         :into RAM location "command"

0287  CD01E1             jsr  TRANSFER        ;Call TRANSFER subroutine to transfer
                                              ;data and command bytes to JCI

028A  81                 rts                  ;Return From Subroutine

                ************************************************************************
                *                                                                      *
                *              Received Message Interrupt Service Routine              *
                *                                                                      *
                ************************************************************************

0300                     org  service

0300  B630               lda  command         ;Save current command byte in
0302  B733               sta  cmdtemp         ;temporary storage location

0304  B632               lda  data            ;Save current data bye in
0306  B735               sta  datatemp        ;temporary storage location

                *** Received Message Interrupt Service Routine ***

0308  A602               lda  #flshbyte       ;Load "flush first byte in FIFO" command
030A  B730               sta  command         ;in command storage location

030C  CD01E1             jsr  TRANSFER        ;Call TRANSFER subroutine, retrieving
                                              ;Status and pri/type data byte.

030F  CD01E1             jsr  TRANSFER        ;Call TRANSFER subroutine, retrieving
                                              ;Status and target i.d. data byte

0312  5F                 clrx                 ;Clear X-Register

0313  B632               lda  data            ;Load target i.d. byte into acc.

0315  A100               cmp  #id1            ;Compare target i.d. with first message
0317  2712               beq  getmsg          ;buffer i.d., if match, get message

0319  5C                 incx                 ;Increment X-Register

031A  A120               cmp  #id2            ;Compare target i.d. with next message
031C  270D               beq  getmsg          ;buffer i.d., if match, get message

031E  5C                 incx                 ;Increment X-Register
```

```
031F  A104                cmp   #id3            ;Compare target i.d. with next message
0321  2708                beq   getmsg          ;buffer i.d., if match, get message

0323  5C                  incx                  ;Increment X-Register

0324  A124                cmp   #id4            ;Compare target i.d. with next message
0326  2703                beq   getmsg          ;buffer i.d., if match, get message

0328  CC0340              jmp   dump            ;Not interested in this message

032B  EE43      getmsg:   ldx   msgl,x          ;Load pointer to corresponding message RAM
                                                ;buffer into X-Register

032D  CD01E1              jsr   TRANSFER        ;Call TRANSFER subroutine, retrieving
                                                ;Status and source i.d. data byte

0330  0E3110    rxdata:   brset rfifoa,status,finish  ;Was previous byte "last byte"
                                                      ;If so, don't load any data

0333  0D310D              brclr rfifob,status,finish  ;Again, if no valid data,
                                                      ;end routine

0336  CD01E1              jsr   TRANSFER        ;Call TRANSFER subroutine, retrieving
                                                ;Status and data bytes

0339  B632                lda   data            ;Load received data into acc., then store
033B  F7                  sta   ,x              ;it in next location in message buffer

033C  5C                  incx                  ;Increment X-Register

033D  CC0330              jmp   rxdata          ;Loop back to "rxdata" to check for
                                                ;another data byte

0340  CD027B    dump:     jst   RXFLUSH         ;Flush current Rx FIFO

0343  B633      finish:   lda   cmdtemp         ;Retrieve command byte and
0345  B730                sta   command         ;store in command byte location

0347  B635                lda   datatemp        ;Retrieve data byte and
0349  B732                sta   data            ;store in data byte location

034B  80                  rti                   ;Return from interrupt

                ************************************************************************
                *                                                                      *
                *                 Timer Interrupt Service Routine                      *
                *                                                                      *
                ************************************************************************

0360                      org   timer

0360  B630                lda   command         ;Store current command byte in
0362  B733                sta   cmdtemp         ;temporary storage location

0364  B632                lda   data            ;Store current data byte in
0366  B735                sta   datatemp        ;temporary storage location
```

```
                 *** Timer Interrupt Service Routine ***

0368  CD0233               jsr   TXSTATUS         ;Call TXSTATUS subroutine

036B  B633                 lda   cmdtemp          ;Retrieve command byte and
036D  B730                 sta   command          ;store in command byte location

036F  B635                 lda   datatemp         ;Retrieve data byte and
0371  B732                 sta   data             ;store in data byte location

0373  80                   rti                    ;Return from interrupt

                 ************************************************************************
                 ***                    MC68HC705C8 Reset Vectors                   *****
                 ************************************************************************

1FF4                       org   vectors

1FF4  0000                 fdb   none             ;SPI
1FF6  0000                 fdb   none             ;SCI
1FF8  0360                 fdb   timer            ;Timer
1FFA  0300                 fdb   service          ;external int. vector
1FFC  0000                 fdb   none             ;software int. vector
1FFE  0180                 fdb   rom              ;reset vector

                 ************************************************************************
                 ***          End of MC68HC705C8/JCI Sample Driver Routines         *****
                 ************************************************************************
```

[redacted]

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

AN1212/D