

# MOTOROLA SEMICONDUCTOR APPLICATION NOTE

**AN1240**

## HC05 MCU Software-Driven Asynchronous Serial Communication Techniques Using the MC68HC705J1A

By Scott George  
CSIC MCU Product Engineering

### INTRODUCTION

This application note describes a method for asynchronous serial communication with a microcontroller unit (MCU) using standard input/output (I/O) port pins and software which incorporate noise and frame-error detection. If error detection is not needed, the code size may be reduced for more efficient use of memory.

### OVERVIEW

A serial communication interface (SCI) is a serial I/O sub system available with many Motorola MCUs. This hardware module provides full-duplex, universal asynchronous receiver/transmitter-type (UART) serial communication between the MCU and other UART-type devices, such as a cathode-ray-tube (CRT) terminal, personal computer, or other MCUs. The SCI handles all transmission and reception duties and by so doing off-loads the CPU to perform other functions simultaneously. The SCI is software programmable for many different baud rates. The receiver can detect error conditions automatically, such as framing, noise, and overrun.

Some Motorola MCUs do not include an SCI, specifically a low-cost, low-pin-count MCU such as the MC68HC705J1A. To perform asynchronous serial communication, software must be used to emulate an SCI. In this case, the CPU would control I/O port pins to perform the same functions as the receive data (RXD) and transmit data (TXD) pins of a true hardware-driven SCI.

This application's software solution requirements are:

- Speed optimization for maximum baud rate
- Minimal code size
- Easy configuration for different baud rates
- Ability to detect noise and framing errors while receiving.

Because the CPU is not as efficient as a dedicated hardware SCI, software emulation has limitations:

- Very fast baud rates are not attainable
- SCI software consumes memory space and CPU bandwidth
- Flexibility and features are reduced



If a particular application cannot be limited by these restrictions, then using an MCU with an SCI would be appropriate. However, many applications do not need the performance or flexibility of an SCI, and, in those cases, software emulation is a cost-effective solution.

The above requirements would be jeopardized by software emulation of full-duplex transmission. This software solution only operates in half-duplex mode.

## SERIAL COMMUNICATION TERMINOLOGY AND CONCEPTS

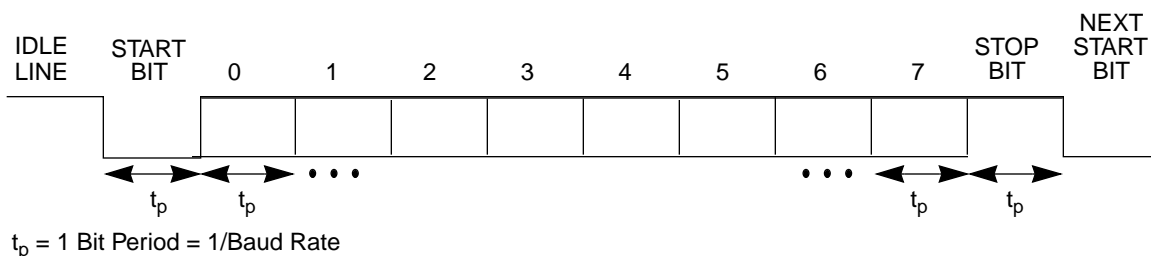
Several technical concepts and terms pertaining to SCI software operation are discussed here. Note that message protocol is not discussed, since it is assumed the reader is knowledgeable about effective SCI communication.

### Half-Duplex Operation

In a half-duplex system, only one node transmits at any one time. The MCU cannot receive while it is transmitting, and it cannot transmit while it is receiving. This inability is in contrast to the hardware SCI, which can transmit and receive different information at the same time. This is known as a full-duplex system.

### Transmission Format

The SCI uses the standard non-return-to-zero (NRZ) format consisting of one start bit followed by one byte (eight bits) of data and one stop bit. This is commonly referred to as an 8-N-1 format (8 data bits, no parity bit, 1 stop bit). Data is both transmitted and received least significant bit (LSB) first. Each bit has a duration,  $t_p$ , which defines the baud rate.

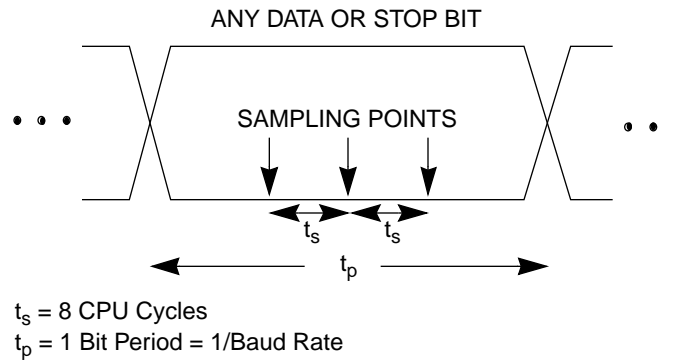


**Figure 1. NRZ 8-N-1 Transmission Format**

As shown in Figure 1, an idle line is high (logic one) prior to transmission or reception, and the start bit is low (logic zero). Each data bit is either high (logic one) or low (logic zero). The stop bit is high (logic one). The start bit, eight data bits, and stop bit constitute one frame of data.

### Noise Detection

On an asynchronous serial network, data transmitted by one node may be received incorrectly by another node because of noise corruption along the data path. To minimize noise corruption, the SCI receiver software routine samples each bit three times in the middle of each bit period (see Figure 2).



**Figure 2. SCI Receiver Sample Points**

The true bit data is derived by the receiver by using a majority rule of the three samples. A noise condition occurs when the three samples are not identical. The SCI receiver software routine sets the half-carry bit to signal a noise condition.

## Frame Error Detection

The stop bit is defined as a logic one. If the stop bit is received as a logic zero, a frame error has occurred. The SCI receiver software routine uses the carry bit to signal a frame-error condition.

# APPLICATION

## System Overview

The application of the SCI software consists of an RS232-C physical interface connecting an MCU to a dumb terminal. As each character is typed on the terminal's keyboard, the ASCII-equivalent data is transmitted to the MCU. The MCU then transmits the ASCII character back to the dumb terminal. If a noise or frame error occurs during the reception of the character, the appropriate LEDs are lit to signal the error.

## Hardware Description

The Motorola MC68HC705J1A MCU and the Motorola MC145407 RS232-C transmitter/receiver are used in this example (refer to Appendix A). The Motorola MC34064 low-voltage reset is connected to the reset pin to provide brown-out and slow supply power-on protection. A ribbon cable connects the MC145407 to the dumb terminal. A 4.0-MHz crystal oscillator clocks the MCU, and both the dumb terminal and the SCI receiver routine are configured for 9600 baud. Other selectable baud rates also may be used.

## Software Description

The SCI software consists of two main subroutines to be called by the main program. The receive routine, **get\_char**, receives one byte of data from the receive data pin (RXD) and places it into **char**, a variable in zero-page RAM. The **get\_char** routine calls a subroutine, **get\_bit**, which captures three samples of the state of RXD and adds them together to derive bit data and noise information. Upon exiting **get\_char**, the

carry bit is set if a noise condition occurred; otherwise, it is cleared. The half-carry bit is set if a frame error occurred; otherwise, it is cleared. **Char** contains the received data.

The transmit routine, **put\_char**, transmits serially the contents of **char** using the transmit data pin (TXD).

Both **get\_char** and **put\_char** call **delay\_13a**, a subroutine which produces a delay of  $13 \times \text{ACC} + 12$  CPU cycles, where ACC is the value in the accumulator at the time the subroutine is called. See Appendix B for flowcharts and Appendix C for the source code listing.

The baud rate for both the receiver and transmitter is selected by changing **BAUD\_SEL** to 4, 8, 16, 32, 64, or 128 which, with a 4.0-MHz crystal oscillator, produces a baud rate of 19.2 k, 9600, 4800, 2400, 1200 or 600 respectively. The baud rate for the receiver and the transmitter will be the same. Appendix D specifies receiver tolerances and transmitter accuracies for each baud rate.

## CUSTOMIZATION

This section introduces possible customization of the software SCI concept. Detailed description of these ideas is beyond the scope of this application note.

### Wake-up and Time-out Features

Wake-up capability of the receiver routine allows the CPU to execute useful code while the RXD line is idle. Both the RXD pin and the IRQ pin are connected to the RXD line. A negative transition on the RXD line will cause an IRQ interrupt. The interrupt service routine can then call **get\_char**. An excellent way to generate a negative transition on the RXD line is to transmit a zero (\$00) immediately followed by the stream of data to be received. Note that the zero is not received, but the data following the zero is received.

Time-out capability of the receiver routine allows an interrupt to abort an idle line condition. Before the **get\_char** routine is called, the multifunction timer (MFT) can be configured to interrupt after a time longer than the anticipated receive time. Care should be taken as to how the subroutine is entered and exited. Note that stack pointer housekeeping might be required.

### Low Voltage Reset Circuitry

An MC34064 low-voltage reset device has been included to show the most robust reset circuit. This provides protection from slow-ramping power supplies. Many bench-type power supplies ramp slowly, causing faulty power-on of MCUs. The MC34064 holds the RESET pin low until the power supply is within a specified range. This also provides protection from brownout, when the MCUs minimum  $V_{DD}$  requirements are exceeded. If such robust protection is not required, engineering judgment may be used to design a more cost-effective circuit.

### Code Minimization

Code size may be minimized by eliminating code specific to noise detection if that feature is not needed in an application. This could result in up to a 30% reduction of code space.

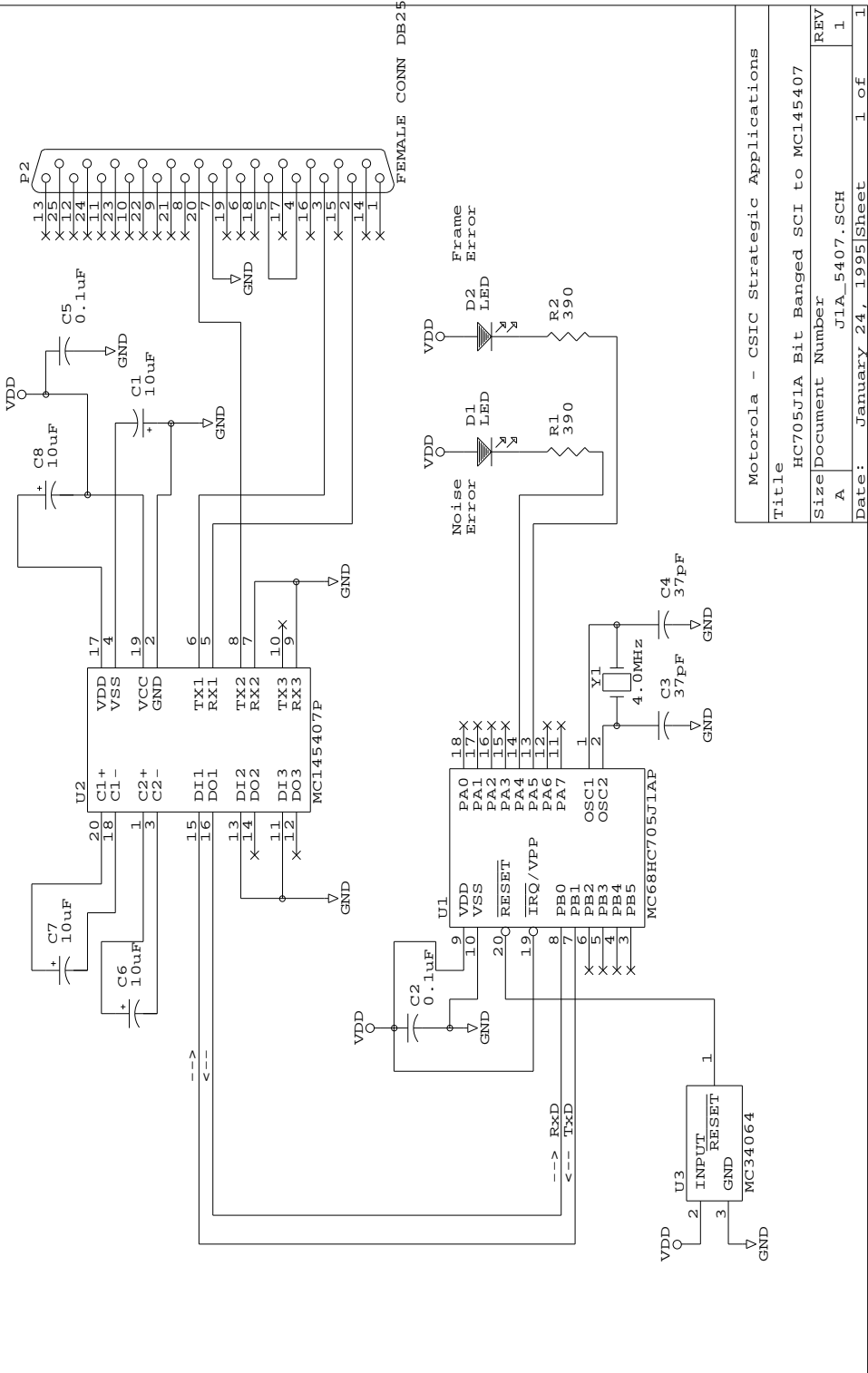
## CONCLUSION

SCI receiver and transmitter software routines offer the application designer an alternative to using a hardware SCI. The software routine listings contain the operational details. The routines may be used as listed or customized as determined by engineering requirements.

An electronic listing of the source code in Appendix C can be found on the Motorola MCU BBS. The BBS phone number is (512) 891-3733. The file name is J1A\_5407.ARC and can be found on the CSIC BBS under the APPNOTES directory.

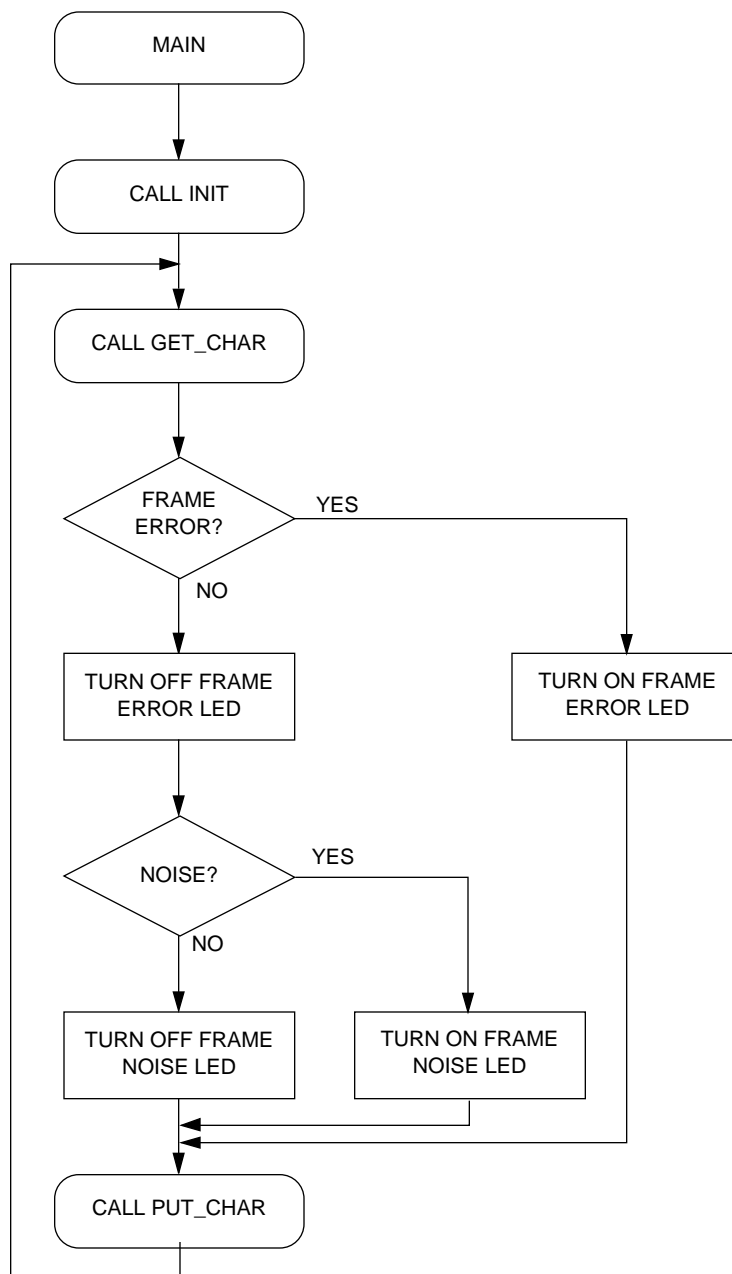
## APPENDIX A

HC705J1A to MC145407 Interface Circuit

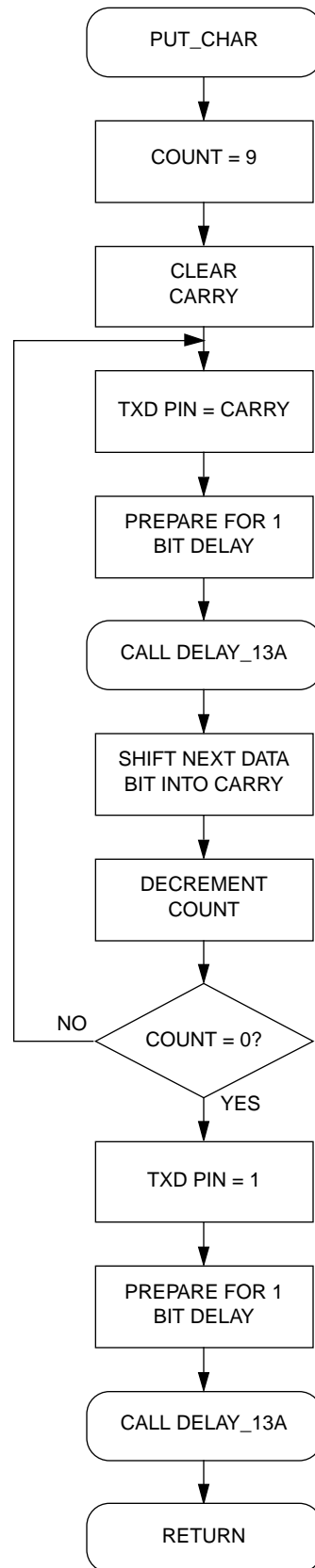
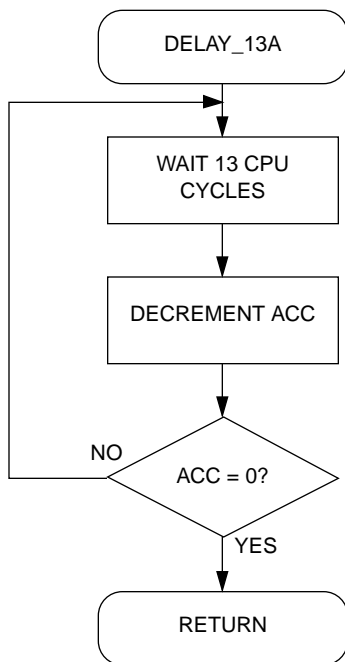
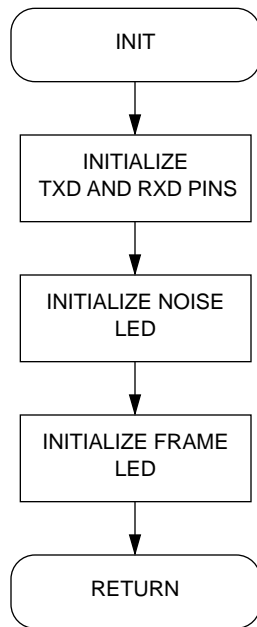


Motorola - CSIC Strategic Applications	
Title	HC705J1A Bit Banged SCI to MC145407
Size	Document Number
A	J1A_5407.SCH
REV	1
Date:	January 24, 1995
Sheet	1 of 1

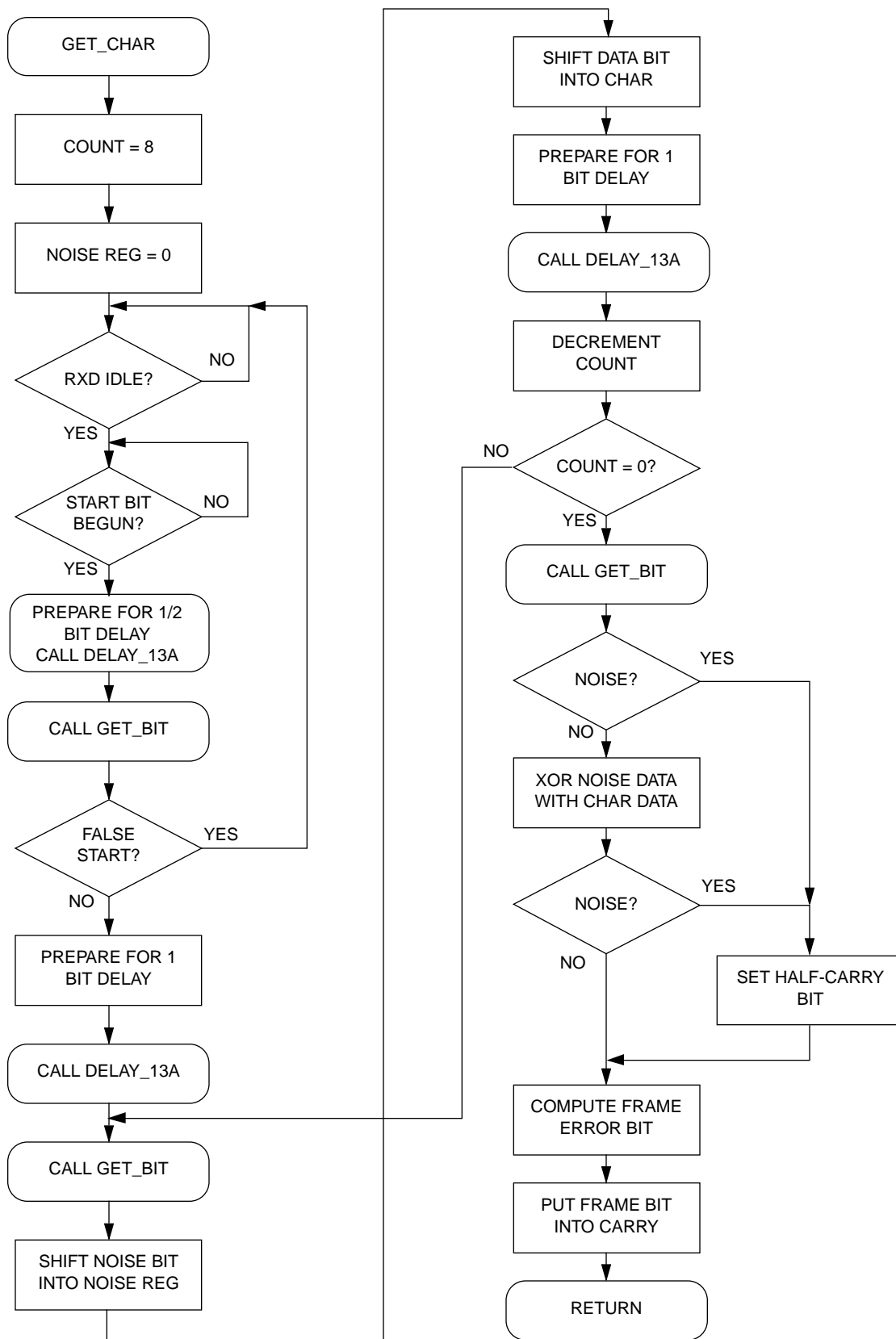
## APPENDIX B

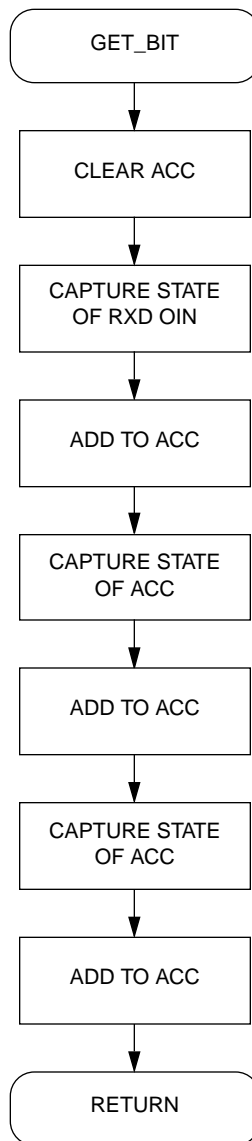












## APPENDIX C

```
*****
*****
*
* Main Routine SCI_01 - SCI Software Transmit/Receive Routines
*
*****
*
* File Name: SCI_01.RTN                      Copyright (c) Motorola 1995
*
* Full Functional Description of Routine Design:
*   Program flow:
*       Reset:  Call init to initialize port pins
*               Call get_char to receive a byte of data
*               Light frame error LED if frame error occurred
*               Light noise LED if frame error occurred
*               Call put_char to transmit the received byte of data
*               Loop back to get_char call (endless loop)
*
*****
*
*   Part Specific Framework Includes Section
*
*****
#include 'H705J1A.FRK'                      ; Include the equates for the
                                           ; HC705J1A so all labels can
                                           ; be found.
```

```

*****
*
*      MOR Bytes Definitions for the Main Routine
*
*****

        org      MOR
        fcb      $20

*****
*
*      Equates and RAM Storage
*
*****

*** I/O Pin Equates:
serial_port      equ      $01          ; port used for serial port
                                   ; pins
status_port      equ      $00          ; port used for driving LED's.
noise            equ      4            ; pin # for noise LED
frame            equ      5            ; pin # for frame LED
rx              equ      0            ; pin # for receive data pin
tx              equ      1            ; pin # for transmit data pin

*** Program Constant Equates:
                                   ; Baud rate select table:
BAUD_SEL          equ      $08          ; BAUD_SEL      4MHz osc   2MHz osc
                                   ;   $04      19.2k      9600
                                   ;   $08      9600      4800
                                   ;   $10      4800      2400
                                   ;   $20      2400      1200
                                   ;   $40      1200      600
                                   ;   $80      600       300

*** RAM variable allocation:
        org      RAM
char          rmb      1              ; data register for sci
count         rmb      1              ; temp storage variable

```

```

*****
* main - example program that continually echoes back received characters. *
*
* input cond.      - reset
* output cond.     - none (infinite loop)
* stack used       - 4 bytes
* variables used   - none
* ROM used         - 28 bytes
*****

                org      ROM                ; start at the top of ROM
main            rsp                ; reset the stack pointer
                jsr      init            ; initialize port pins

main_loop       jsr      get_char        ; receive one byte of data
                                           ; from rxd pin

                bcc      no_frame_error  ; branch if no noise occurred
                bclr     frame,status_port ; turn on frame LED
                bra      continue        ; don't check for noise --
                                           ; it's undefined
no_frame_err    bset      frame,status_port ; turn off frame LED

                bhcs     noise_error     ; branch if noise occurred
                bset     noise,status_port ; turn off noise LED
                bra      continue        ; skip next line of code
yes_noise_err   bclr     noise,status_port ; turn on noise LED

continue        jsr      put_char        ; transmit the received byte
                bra      main_loop       ; and prepare for next
                                           ; reception.

```

```

*****
* init - initialize port pins for sci operation and for driving LEDs;      *
*      called by main                                                       *
*                                                                           *
* input cond.      - none                                                  *
* output cond.     - TXD = output initialize to 1, RXD = input, noise LED = *
*                   off, frame LED = off.                                  *
* stack used       - 0 bytes                                               *
* variables used   - none                                                 *
* ROM used         - 15 bytes                                              *
*****

init          bset      txd,serial_port      ; init txd = 1
              bset      txd,serial_port+4    ; txd = output

              bclr      rxd,serial_port+4    ; rxd = input

              bset      noise,status_port    ; noise LED = off
              bset      noise,status_port+4  ; noise = output

              bset      frame,status_port    ; frame LED = off
              bset      frame,status_port+4  ; frame = output
              rts                      ; exit (init)

*****

* get_char - receive one byte of data from RXD pin; called by main        *
*                                                                           *
* input cond.      - RXD pin defined as an input pin                      *
* output cond.     - char contains received data; X,ACC undefined;        *
*                   half carry = 1 (frame occurred) or 0 (no frame error); *
*                   carry = 1 (noise and/or frame error occurred) or 0    *
*                   (no noise).                                            *
* stack used       - 2 bytes                                               *
* variables used   - char: storage for received data (1 byte)            *
*                   count: temporary storage (1 byte)                    *
* ROM used         - 63 bytes                                              *
*****

```



get_char	lda	#8	;	[2]	receiving 8 data bits	
	sta	count	;	[4]	store value into RAM	
	clr	rx	;	[3]	used to store noise data	
get_start_bit	brclr	rx,serial_port,*	;	[5]	wait until rx=1	
	brset	rx,serial_port,*	;	[5]	wait for start bit	
	lda	#BAUD_SEL-3	;	[2]	prepare for 1/2 bit delay	
	bsr	delay_13a	;	[13a+12]	execute delay routine	
	bsr	get_bit	;	[39]	sample start bit	
	lsra		;	[3]	noise bit -> carry;	
			;		acc=filtered start bit	
	bne	get_start_bit	;	[3]	if false start, start over	
	tsta		;	[3]	for timing purposes only	
	tsta		;	[3]	for timing purposes only	
	lda	#2*(BAUD_SEL-2)	;	[2]	prepare for 1 bit delay	
	bsr	delay_13a	;	[13a+12]	execute delay routine	
	get_data_bits	bsr	get_bit	;	[39]	sample data bit
		rora		;	[3]	noise bit -> carry
rorx			;	[3]	carry -> noise data reg	
rora			;	[3]	filtered data bit -> carry	
ror		char	;	[5]	carry -> char	
lda		#2*(BAUD_SEL-3)	;	[2]	prepare for 1 bit delay	
bsr		delay_13a	;	[13a+12]	execute delay routine	
tsta			;	[3]	for timing purposes only	
dec		count	;	[5]	bit received, dec count	
bne		get_data_bits	;	[3]	loop if more bits to get	
get_stop_bit		bsr	get_bit	;	[39]	sample stop bit
		lsra		;	[3]	noise bit -> carry
			;		acc=filtered stop bit	
	sta	count	;	[4]	store stop bit in count	
	bcc	yes_noise	;	[3]	if noise, then branch	

```

                                txa                ;[2] noise data -> acc
                                eor      char        ;[3] XOR noise with char,
                                beq      no_noise     ;[3] and if result=0,
                                                ;    then no noise in data
                                                ;    reception

yes_noise      lda      #$08                ;[2] set noise bit (half carry)
                                add      #$08                ;[2] by adding $8 to $8

no_noise       lda      count                ;[3] retrieve stop data bit,
                                coma                ;[3] complement it,
                                lsra                ;[3] and shift it into carry
                                                ;    for frame error bit

                                rts                ;[6] exit (get_char)

*****

* get_bit - receive one bit of filtered data and noise info; called by      *
*      get_char                                                              *
*                                                                            *
* input cond.      - RXD pin defined as an input pin                        *
* output cond.     - ACC = 000000dn, where d = filtered data, n = noise info *
* stack used       - 0 bytes                                                *
* variables used   - none                                                  *
* ROM used         - 17 bytes                                               *
*****

get_bit        clra                ;[3] used to add sampled bits
                                brset    rxd,serial_port,samp_1 ;[5] sample 1st bit into carry
samp_1         adc      #0                ;[3] add it to acc
                                brset    rxd,serial_port,samp_2 ;[5] sample 2nd bit into carry
samp_2         adc      #0                ;[3] add it to acc
                                brset    rxd,serial_port,samp_3 ;[5] sample 3rd bit into carry
samp_3         adc      #0                ;[3] add it to acc
                                rts                ;[6] exit (get_bit)

```

```

*****
* put_char - transmit data byte in char out onto TXD pin; called by main      *
*                                                                              *
* input cond.      - TXD pin defined as an output pin and TXD = 1;          *
*                  char contains byte to be transmitted.                    *
* output cond.     - X,ACC,char = undefined;                                *
* stack used       - 2 bytes                                                *
* variables used   - char: storage for transmitted data (1 byte)            *
* ROM used         - 31 bytes (35 if sending two stop bits)                 *
*****

put_char      ldx      #9                      ;[2] be sending 8 data bits
              clc                      ;[2] clear carry for start bit

put_data_bits bcc      send_0                  ;[3] if carry<>0, then
              bset     txd,serial_port         ;[5]   send out a 1
              bra      jmp_bit                  ;[3]   finished sending a 1
send_0        bclr     txd,serial_port         ;[5] else send a 0
              bra      jmp_bit                  ;[3]   finished sending a 0
jmp_bit       lda      #2*(BAUD_SEL-1)-1       ;[2] prepare for a 1 bit delay
              bsr      delay_13a               ;[13a+12] execute delay routine
              tsta                      ;[3] for timing purposes only
              ror      char                  ;[5] get next data bit to send
              decx                      ;[3] one bit sent, so dec count
              bne      put_data_bits           ;[3] loop if more bits to send

put_stop_bit  nop                      ;[2] for timing purposes only
              bset     txd,serial_port         ;[5] send out a one
              lda      #2*(BAUD_SEL-1)         ;[2] prepare for a 1 bit delay
              bsr      delay_13a               ;[13a+12] execute delay routine

* add the next two lines to guarantee sending two stop bits:
*          lda      #2*(BAUD_SEL-1)+1         ;[2] prepare for a 1 bit delay
*          bsr      delay_13a                 ;[13a+12] execute delay routine

              rts                      ;[6] exit (put_char)

```

```

*****
* delay_13a - delay for 13*ACC + 12 cycles; called by get_char and put_char *
*
* input cond.      - ACC set to appropriate value (13*ACC + 12 cycles) *
* output cond.     - ACC = 0 *
* stack used       - 0 bytes *
* variables used   - none *
* ROM used         - 7 bytes *
*****

```

```

delay_13a      nop                ;[2] this is a 13-cycle loop
               nop                ;[2]
               tsta               ;[3]
               decb               ;[3] decrement loop count
               bne    delay_13a   ;[3] loop if count not zero
               rts                ;[6] exit (delay_13a)


```

```

*****
*
*      Interrupt and Reset vectors for Main Routine
*
*****

               org    RESET
               fdb     main

```

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

#### Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.

EUROPE: Motorola Ltd.; European Literature Centre; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.

JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.

ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No.2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.



**MOTOROLA**

AN1240/D



## APPENDIX D

### Receiver Tolerances

The following tolerances state the maximum variation of the average bit period allowable for accurate reception of data without noise or frame error conditions occurring.

**Table 1 Receiver Tolerances**

Baud Rate for 4 MHz clock (bits/sec)	Baud Rate for 2 MHz clock (bits/sec)	Bit Period $t_p$ ( $\mu$ s)	Bit Period Tolerance
19.2k	n/a	52.08	+2.7%/-4.0%
9600	9600	104.2	+3.7%/-5.7%
4800	4800	208.3	+3.9%/-5.5%
2400	2400	416.7	+4.3%/-4.8%
1200	1200	833.3	+4.9%/-5.2%
600	600	1666.7	+4.9%/-5.4%
n/a	300	3333.3	+4.9%/-5.1%

### Transmitter Accuracy

The following table states the percent accuracy of the transmitted bit period to the ideal bit period.

**Table 2 Transmitter Accuracy**

Baud Rate for 4 MHz clock (bits/sec)	Baud Rate for 2 MHz clock (bits/sec)	Ideal Bit Period $t_p$ ( $\mu$ s)	Actual Bit Period $t_p$ ( $\mu$ s)	% Accuracy
19.2k	n/a	52.08	52.0	0.16%
9600	9600	104.2	104.0	0.16%
4800	4800	208.3	208.0	0.16%
2400	2400	416.7	416.0	0.16%
1200	1200	833.3	832.0	0.16%
600	600	1666.7	1664.0	0.16%
n/a	300	3333.3	3328.0	0.16%

## REFERENCES

- 1) Motorola, *M68HC11 Reference Manual*, Prentice Hall, Englewood Cliffs, New Jersey, 1989, Order no. M68HC11RM/AD.
- 2) Motorola, *M68HC05 Applications Guide*, Revision 1, Order no. M68HC05AG/AD.
- 3) Motorola, *MC68HC05J1A Technical Data*, Order no. M68HC05J1A/D
- 4) Steve Leibson, *The Handbook of Microcomputer Interfacing, Second Edition*, TAB Books, Inc., Blue Ridge Summit, Pennsylvania, 1989.