Application Note

AN2637/D
1/2004

Software SCI for the MC68HC908QT/QY MCU

By: Pavel Lajsner
Freescale Czech System Labs
Roznov p.R., Czech Republic

General Description

Motorola's PC master system provides a method for remotely controlling almost any kind of application imaginable via a graphical user interface.

The system consists of software running on a PC, with a second piece of software embedded in the target application. The PC and the target communicate with each other using a standard method: a PC serial COM port and a MCU SCI port.

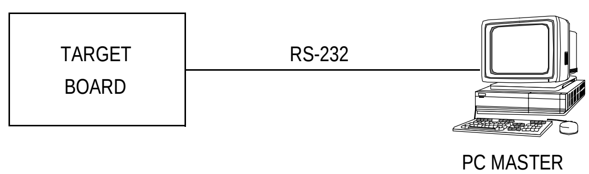


Figure 1. Connection between PC and Target Board

This system can be used for debugging, monitoring, and controlling the target application on-the-fly. It can also be used for intuitive, graphical demonstrations of the target board application functionality.

The embedded application must be ported to platforms (processor) used on the target board. The PC master software remains the same, independent of the target platform. It basically reads and writes the application variables and provides other functions needed for monitoring, controlling, or debugging the target board application.

The PC master software, its usage, protocol, and a few target implementations are described in several Freescale documents and application notes. See the **References** section.

M68HC08 Family and PC Master

The PC master software implementation on SCI-equipped M68HC08 MCUs is straightforward. Because the original embedded core was written in C coding language, the developer must prepare only the initialization and control routines for the M68HC08 SCI.

For M68HC08 MCUs that do not have an available SCI, the serial communication must be provided by means of software. This application note provides an example of a C code implementation of a software SCI for the PC master software. Such a solution has some limitations, which are also discussed in this document.

This software SCI solution can be also used in other applications where the limitations are acceptable. See details in **System Limitations**.

Figure 2 gives an example of the traditional PC master solution (on M68HC08 MCUs with true SCI).

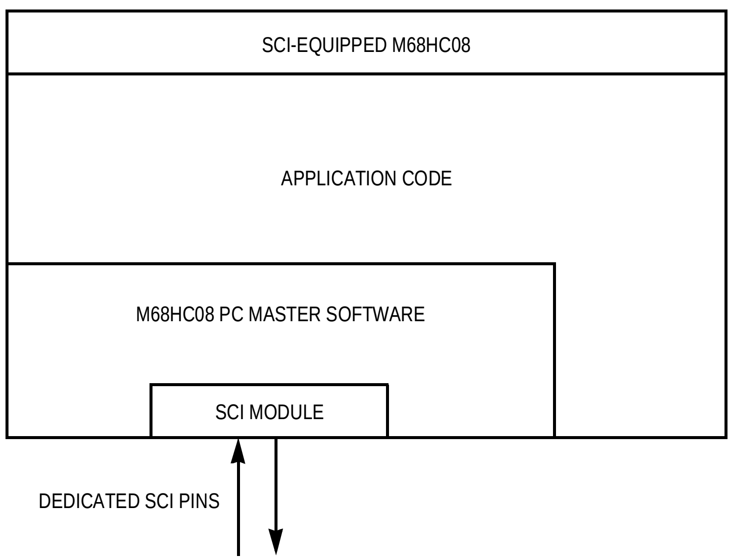


Figure 2. Traditional (SCI-Equipped) M68HC08 PC Master Software

In this scenario, only dedicated SCI pins are occupied by PC master, plus some CPU time is consumed serving SCI communication requests. The application code runs independently of the PC master. Typically, very few restrictions arise within this combination.

PC Master on M68HC08 MCUs That Do Not Have Available SCI

In contrast, the implementation of a software SCI requires more of the CPU resources. This example demonstrates implementing a fully interrupt-driven solution that uses only one channel of the 16-bit M68HC08 timer.

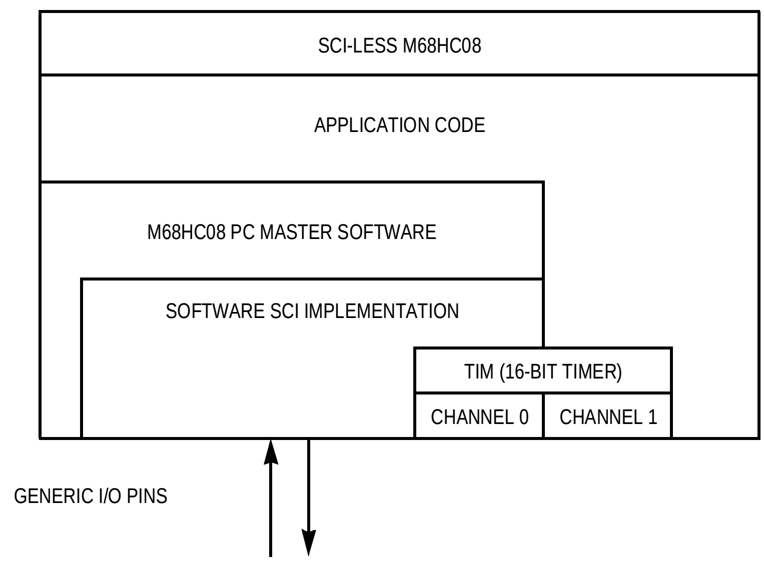


Figure 3. M68HC08 PC Master Software for M68HC08 MCUs That Don't Have SCI

Figure 3 is a relational diagram of the system components. The software implementation was mainly targeted for the lower cost M68HC08 Family members (QT/QY Family), so the main goal was to use the least possible MCU resources.

In addition, a single-wire version of the communication has been developed. The PC master running on the smallest 8-pin QT/QY MCU can be easily demonstrated. This version occupies only one pin and requires minimum internal MCU resources.

Basic 8-Pin CPU PC Master Demo

The software implementation gives a little more freedom, so the usual TTL to RS-232 level-shifting interface can be omitted entirely. Such a solution is perfectly functional on recent motherboards. Here, the RS-232 receivers are formed by Schmitt trigger gates with a threshold voltage of near 1 V, which allows them to be driven by TTL levels (5 V/0 V). Although this doesn't fully conform to the RS-232 specifications, simple, non-critical demo applications may use it. A short cable should be used.

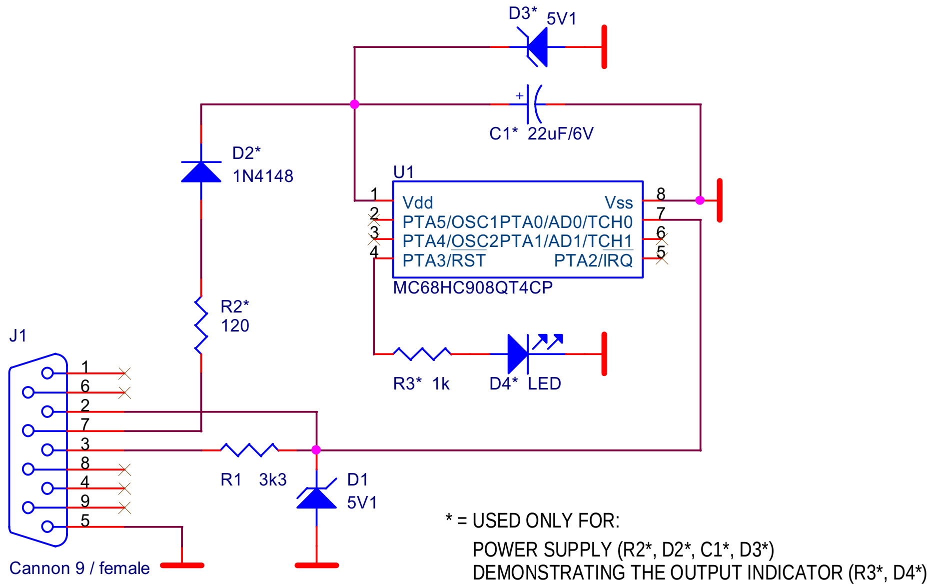


Figure 4. Basic Demo Schematic

Figure 4 shows a very simple configuration of the single-wire communication. Pin 3 (RS-232 output from the PC) goes through a level-limiting circuit (R1, D1) directly to PTA0 (pin 7) of the MCU. The same pin is also used to transmit data from the MCU to the PC (via RS-232 input, pin 2).

All other components (marked by asterisk) are used only for power supply (R2*, D2*, C1*, D3*), or for demonstrating the output indicator (R3*, D4*).

Software SCI Description

This section describes the software SCI routines developed for PC master software to be used on M68HC08 MCUs that do not have an available SCI. Several requirements were defined from the start:

- All routines written in C language
- All processes are fully interrupt driven
- Communication is half-duplex (only one action—receive or transmit—is allowed at a time)
- Uses the least possible CPU resources (ideally, one channel of the 16-bit timer only)

Software Versions

As described above, there are several versions of the software. All features are selected at the compile time by the set of several `#define` directives in `pcmastersoftsci.h` header file.

directive SCISINGLEWIRE

defined: the software will conform to the single-wire communication, the transmit line will go to the third state (allowing reception over the same line)

undefined: the transmit software will behave in the normal way (transmit line will be active all of the time)

directive SCIINV

defined: the SCI communication signal polarity is inverted (idle = 0 V, mark = 5 V). This allows omission of the RS-232 level shifters and inverters (see **Figure 4**)

undefined: regular SCI communication signal polarity is maintained, (idle = 5 V, mark = 0 V), and the RS-232 level shifters are required.

directive SCITXDPINISTIMERPIN

defined: transmit pin uses the output compare feature of 16-bit timer module.

undefined: transmit pin is software controlled, several other define directives are required to define which pin is used:

```
#define TXDPIN       PTA3
#define TXDPINDDR    DDRA_BIT3
#define TXDPINPUE    PTAPUE_BIT3
```

directive SCIRXDPINISTIMERPIN

defined: receive pin uses the input capture feature of 16-bit timer module.

undefined: receive pin is software controlled; several other define directives are required to define which pin is used:

```
#define RXDPIN       PTA3
#define RXDPINDDR    DDRA_BIT3
#define RXDPINPORT   PTA
#define RXDPINMASK   0x08
```

In addition, one more define specifies that the KBI feature of a respective pin is used and what its number (name) is:

```
#define KBIECH KBIER_KBIE3
```

Because KBI can detect only the falling edge, this version cannot be used together with the SCI signal inversion (SCIINV).

Receive Pin

Because one version of the software SCI implementation uses the 16-bit timer, the timer's input capture feature is used to detect the start bit of serial communication. If using this version, the receive pin must be on the timer pin.

Another version of the software was also developed to provide an alternative receive pin option. It uses the QT/QY Family's keyboard interrupt (KBI) module, which is able to detect a falling edge (idle to mark transition, start bit), as the receive pin.

With this version of the software, the receive pin must be on one of the following:

- Timer pin
- Any pin that is KBI capable (all port A pins on QT/QY Family)

Transmit Pin

The selection of the transmit pin is less critical, and there are two options. If the transmit pin is also the timer pin, the output compare feature of the 16-bit timer module can be used, thus the edge generation is precise. This is the preferred solution.

Otherwise, the transmit pin can be any I/O pin since it can be software-driven by the timer interrupt routine. It has been proven that this version works very well too.

The software SCI routines generally use just one timer channel. The selected channel is defined symbolically by the following define directives:

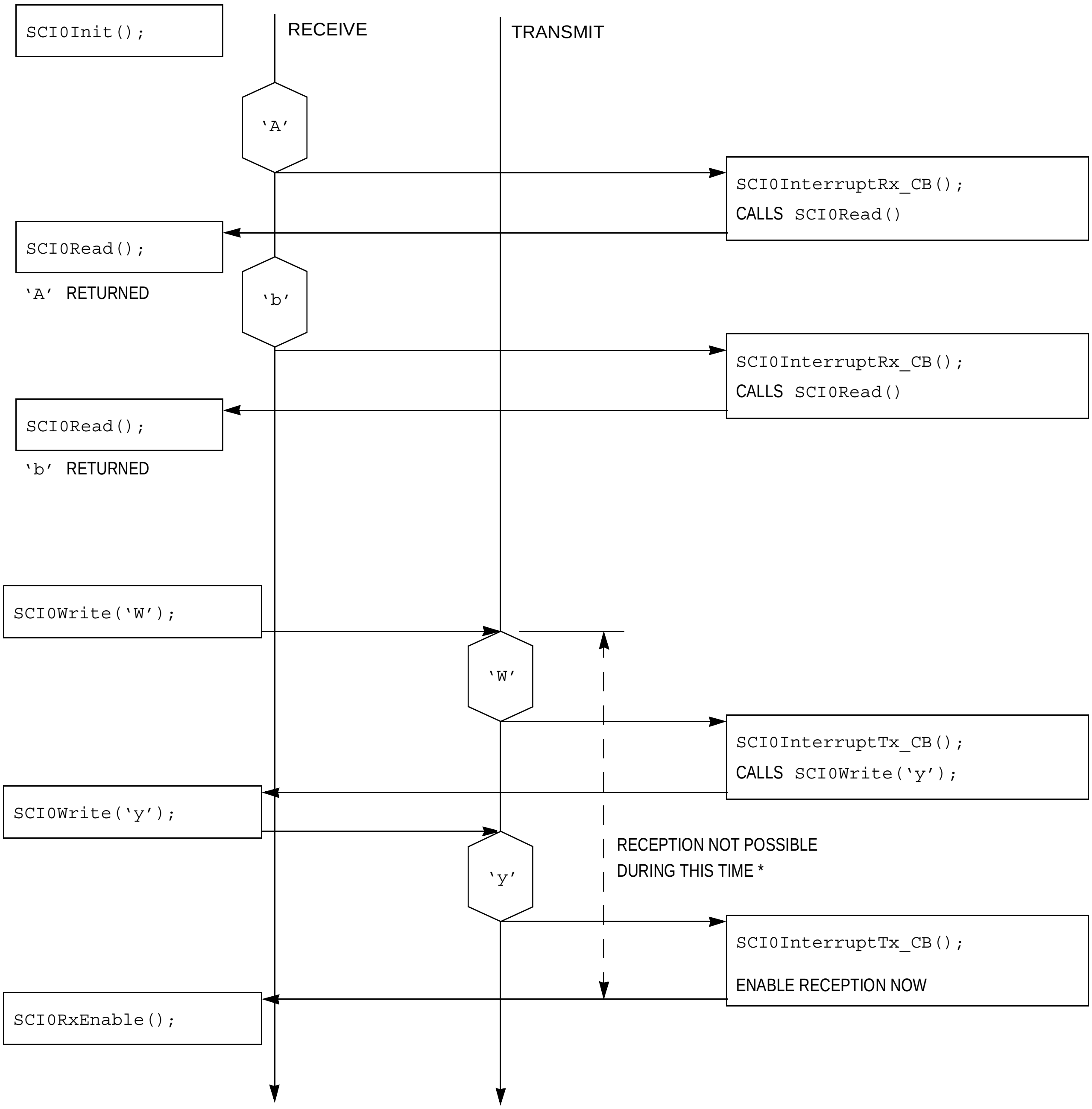
```
#define SCITSC      TSC
#define SCITSCCH    TSC0
#define SCITSC_CHF  TSC0_CH0F
#define SCITSC_IE   TSC0_CH0IE
#define SCITCNT     TCNT
#define SCITCH      TCH0
#define SCITMOD     TMOD
#define IV_SCITMR   IV_TCHO
```

Software SCI API

The software SCI routines communicate with the application program over several functions that together create an API (application program interface). These functions are declared in `pcmastersoftsci.h` header file:

Table 1. Software SCI API Functions

Function	Description
`void SCI0Init(void);`	This function must be called at application start. It will initialize the necessary variables and timers. The SCI reception will be enabled on exiting this routine. The standard settings (9600 bps baud rate, 8 bits, no parity) is used for PC master communication.
`void SCI0Write(char ch);`	Calling this routine will initiate SCI transmission of the character. No checks are made on whether the SCI transmitter is empty or whether SCI reception is in progress.
`void SCI0InterruptTx_CB(void);`	This is a call-back function that must be defined in the user application. Only one source of transmit interrupt is currently implemented — ‘Transmitter Empty’ condition, meaning that a new character can be transmitted. When the current transmission of a character is finished, this function is called by the SCI and `SCI0Write()` can be called again.
`void SCI0InterruptRx_CB(void);`	This is a call-back function that must be defined in the user application. Only one source of receive interrupt is implemented — ‘Receiver Buffer Full’ condition, meaning that a new character was received and must be fetched by the application. This is done by the calling `SCI0Read()` function.
`char SCI0Read(void);`	This function returns the SCI value last received. It must be called after being signalled by `SCI0InterruptRx_CB()` function but before the next character is fully received. Otherwise, the previous value in the receive buffer is overwritten and lost.
`void SCI0RxEnable(void);`	This auxiliary function simply re-establishes reception (usually after the transmission is finished). Any transmission or reception in progress is aborted.



* RECEPTION NOT POSSIBLE BECAUSE ONLY ONE TIMER IS SHARED BETWEEN TRANSMISSION AND RECEPTION

Figure 5. Software SCI API Usage

Detailed Software Description

This section provides a detailed description of all software versions.

Transmission

Output Compare Driven Transmit

This version of the transmit routine uses the output compare (OC) feature of the 16-bit timer (hardware sets a pre-defined level on timer output at a pre-defined time). This provides the precise timing for the transmit signal which is fully determined by the 16-bit timer hardware and independent of any process that could delay the generation of software SCI signals.

The transmission starts with `SCI0Write()` routine, which initializes the timer output compare, to generate a falling edge (as the start bit condition, idle to mark transition) and enables the timer interrupt.

All subsequent events are interrupt driven. The timer hardware sets out the proper level on the timer pin and generates the timer interrupt request. The interrupt service routine then configures the timer for the next output compare event. When all bits are sent out, further timer interrupts are disabled and the `SCI0InterruptTx_CB()` call-back is called. In this routine, the user code determines whether more characters are to be sent.

Figure 6 shows the mutual dependencies in the time domain.

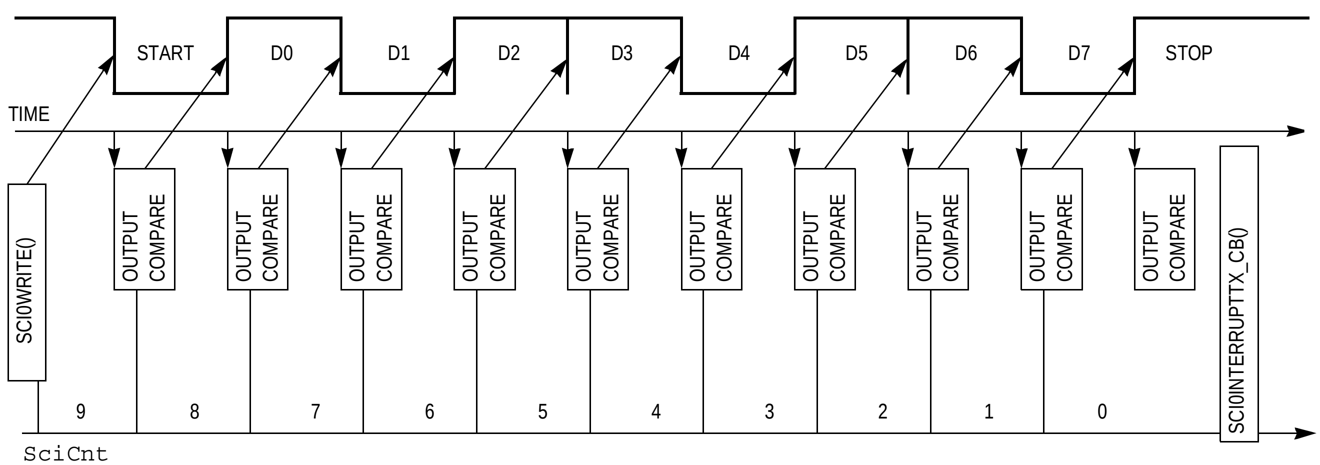


Figure 6. Output Compare Driven Transmission Time Chart

Figure 7 contains flow charts of the routines related to this version of software.

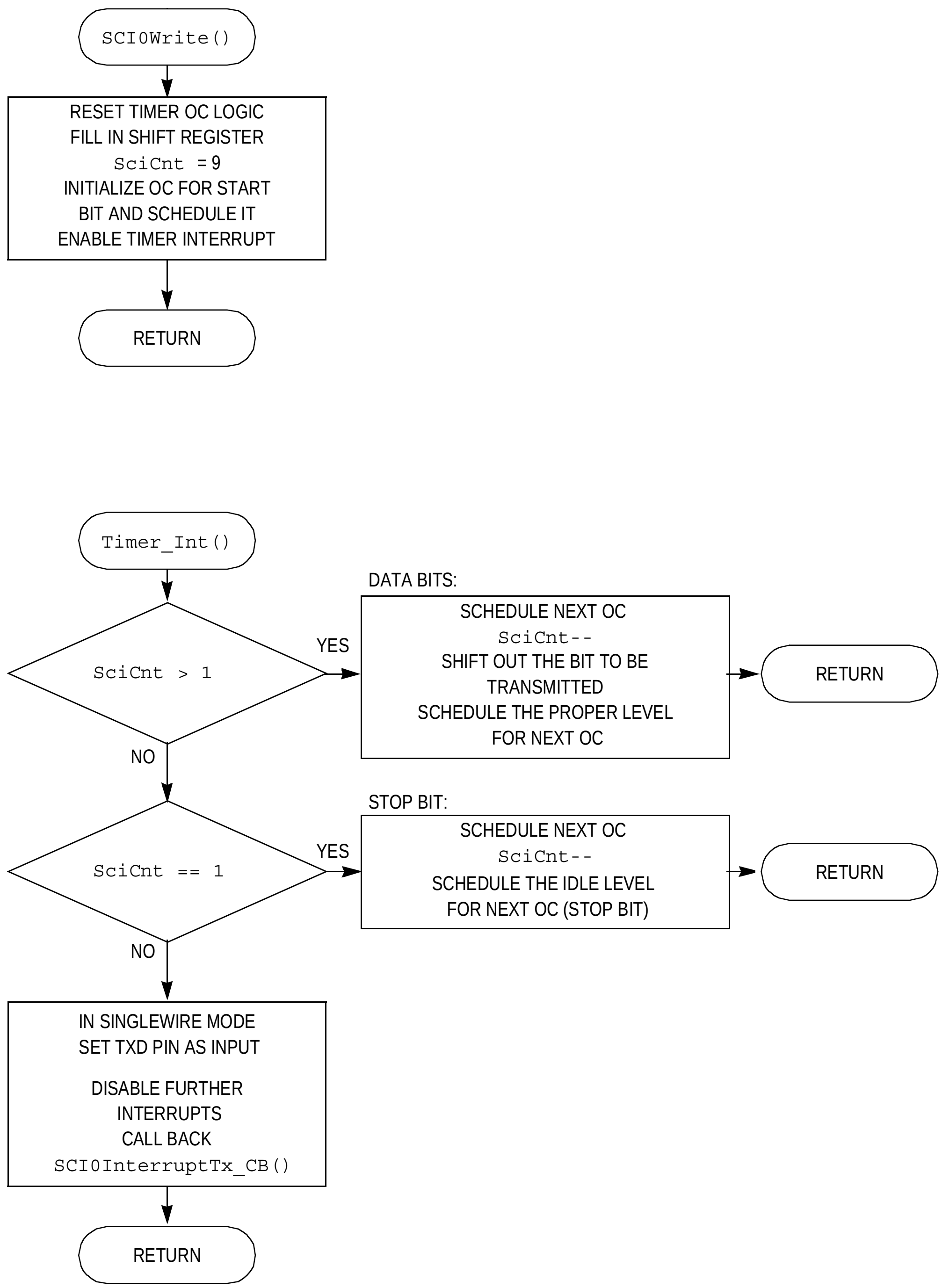


Figure 7. Output Compare Driven Transmit Software Flow Chart

Direct Port Control Transmit

This version of the transmit routine does **not** use pin hardware control features of the 16-bit timer. Actual control of the transmit line is through software, using I/O access to the pin dedicated to transmission. This basically allows using any output-capable pin for transmission. The timer is still used to generate the periodic interrupt requests.

The transmission starts with `SCI0Write()` routine, which initializes the timer and directly clears the transmit line to indicate a start bit condition.

All subsequent events are output compare interrupt driven. The timer generates timer interrupt requests. The timer interrupt service routine then sets/clears the transmit line and configures the timer for the next output compare event. Any other interrupt request that has just been processed will delay execution of the timer interrupt service routine, thus also delaying the transmit signals. See details in **System Limitations**.

When all bits are sent out, further interrupts are disabled and the `SCI0InterruptTx_CB()` call-back is called. In this routine, the user code determines whether more characters are to be sent.

Figure 8 shows the mutual dependencies in the time domain.

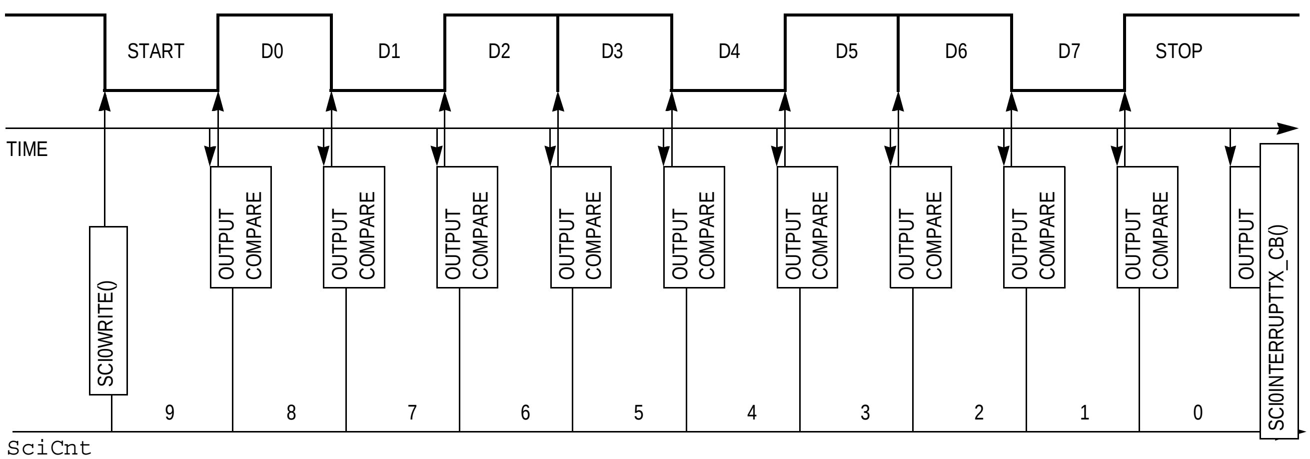


Figure 8. Direct Port Control Transmission Time Chart

Figure 9 contains the flow charts of routines related to this version of software.

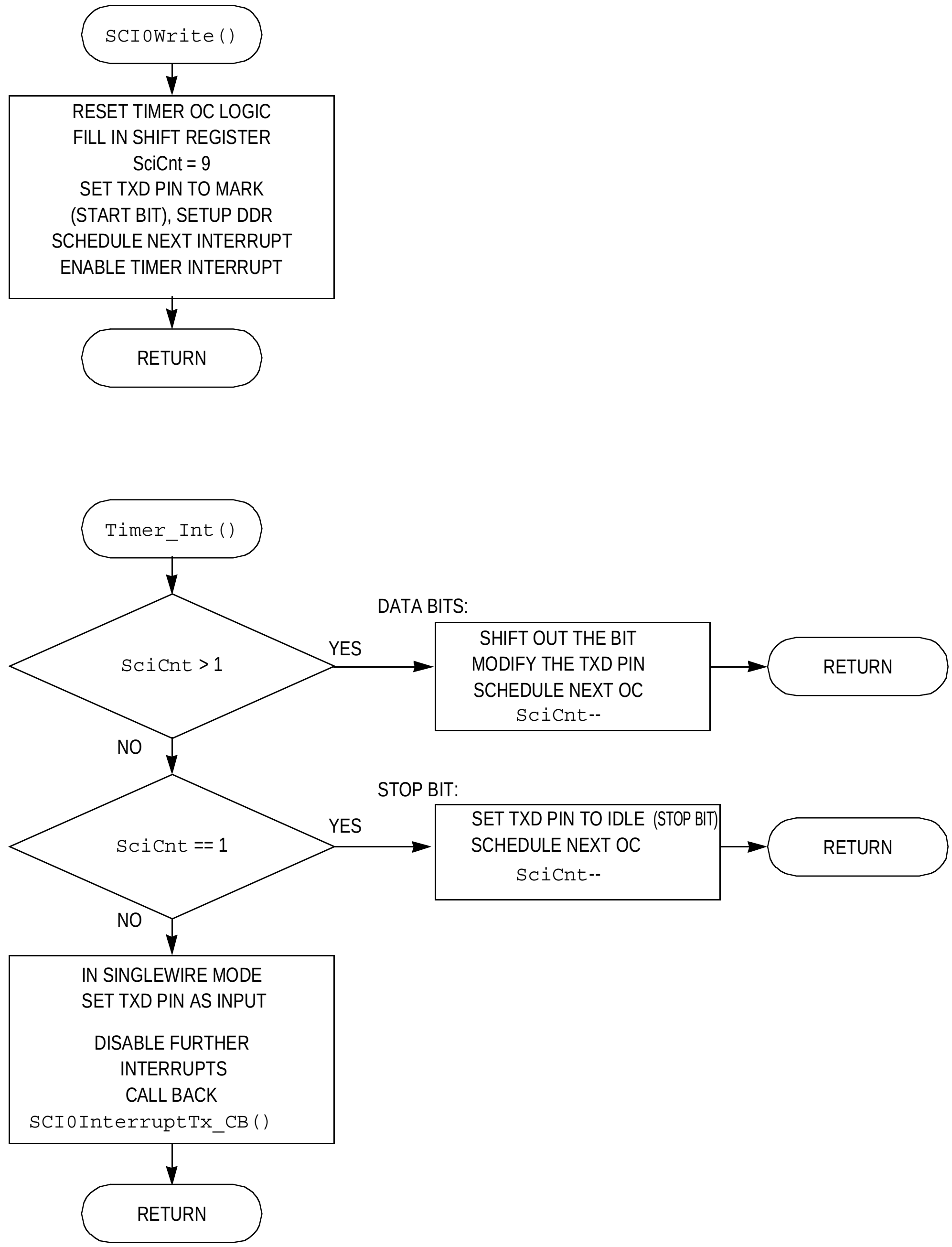


Figure 9. Direct Port Control Transmit Software Flow Chart

Reception

Input Capture Start Bit Detection

This version of the receive routine uses the input capture hardware feature of the 16-bit timer to a detect start bit condition (falling edge, idle) to mark transition. This limits the selection of the receive pin to the timer pins only. Reception may start after the receive software and hardware is initialized using `SCI0RxEnable()` function.

The actual reception starts with a falling edge on the receive line that generates the input capture interrupt. In the input capture interrupt service routine, the timer is reconfigured to generate output compare events.

All subsequent events are driven by periodic output compare interrupts. The timer interrupt service routine then reads the level on the receive line. Any other interrupt request that has just been processed will delay execution of the timer interrupt request routine, thus also delaying the reading of receive signals. See details in **System Limitations**.

When all bits have been received, reception is re-initialized using `SCI0RxEnable()` function. Then the `SCI0InterruptRx_CB()` call-back is called. In this routine, the user code should read the data that was actually received.

Figure 10 shows the mutual dependencies in the time domain.

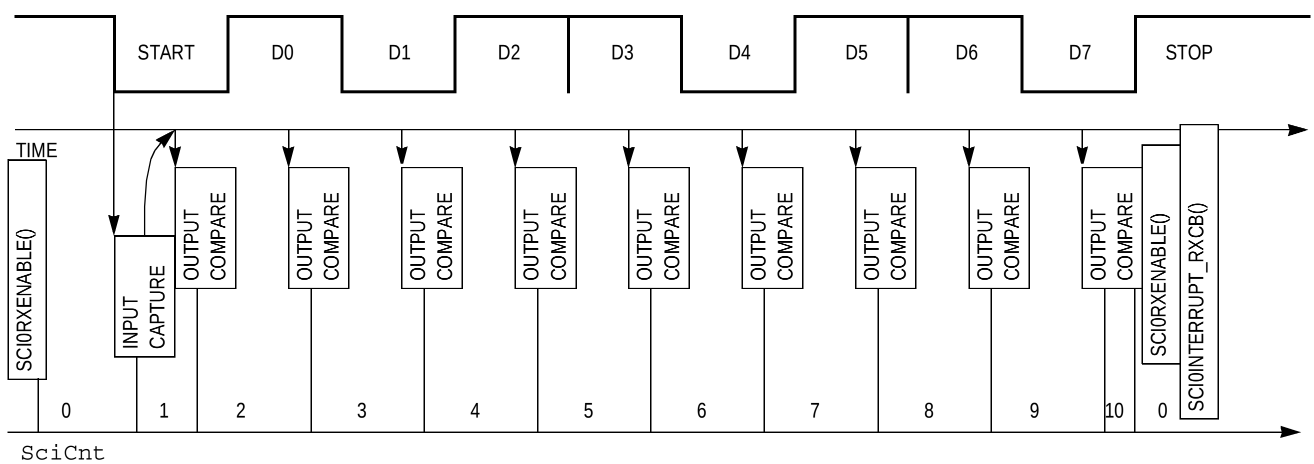


Figure 10. Input Capture Start Bit Detection Software Time Chart

Figure 11 contains the flow charts of routines related to this version of software.

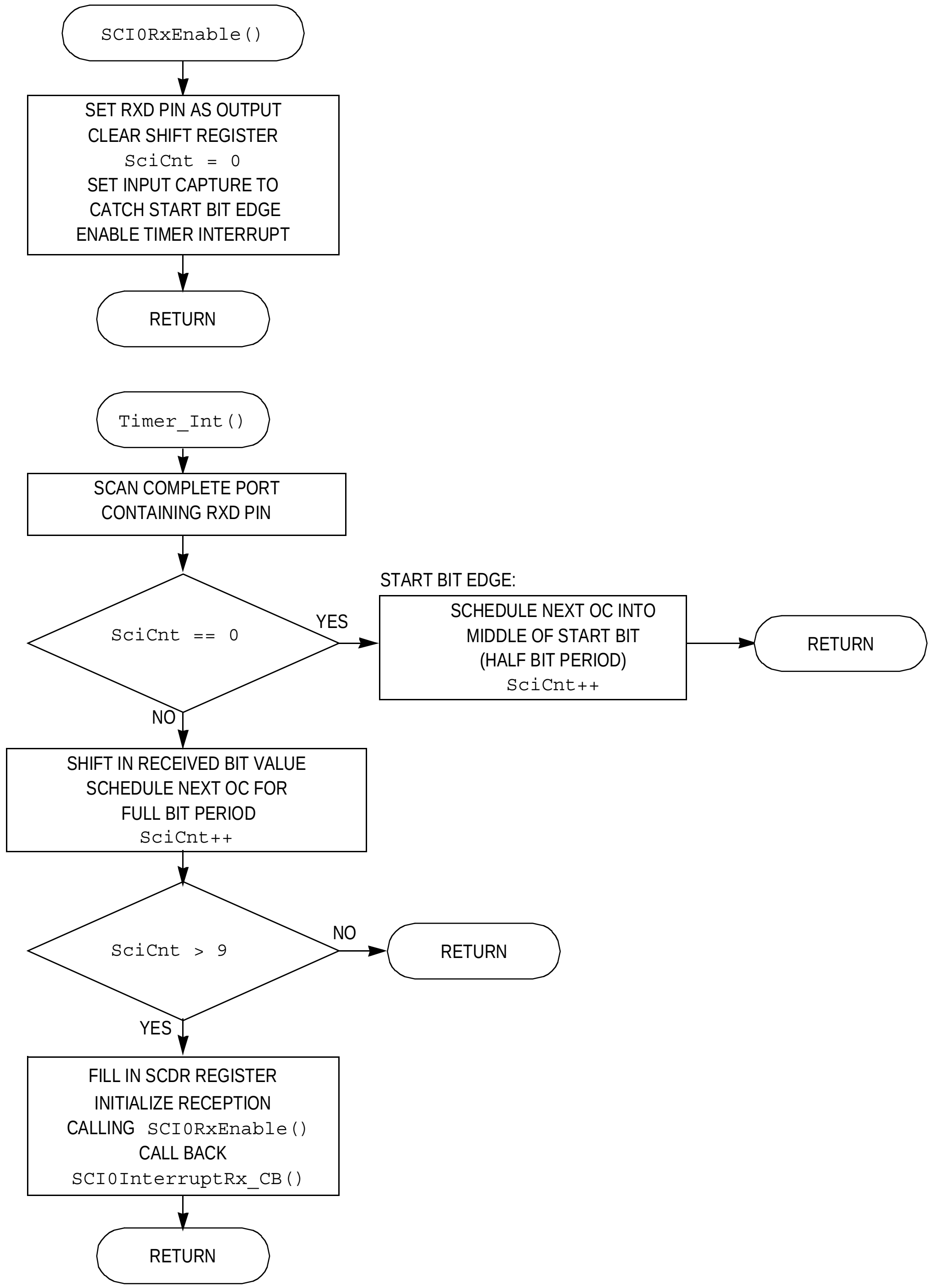


Figure 11. Input Capture Start Bit Detection Software Flow Chart

Keyboard Interrupt (KBI) Start Bit Detection

This version of the receive routine uses the KBI feature to detect a start bit condition (falling edge, idle to mark transition). This allows the option to use any KBI-capable pin as the receive pin. Reception may start after the receive software and hardware is initialized using the `SCI0RxEnable()` function.

The actual reception starts with a falling edge on the receive line that generates the keyboard interrupt. In the keyboard interrupt service routine, the keyboard interrupt is disabled and the timer is configured to generate output compare events.

All subsequent events are driven by periodic output compare interrupts. The timer interrupt service routine then reads the level on the receive line. Any other interrupt request that has just been processed will delay execution of the timer interrupt request routine, thus also delaying the reading of receive signals. See **System Limitations** for details.

When all bits have been received, reception is re-initialized using the `SCI0RxEnable()` function. Then the `SCI0InterruptRx_CB()` call-back is called. In this routine, the user code should read the data that was actually received.

Figure 12 shows the mutual dependencies in the time domain.

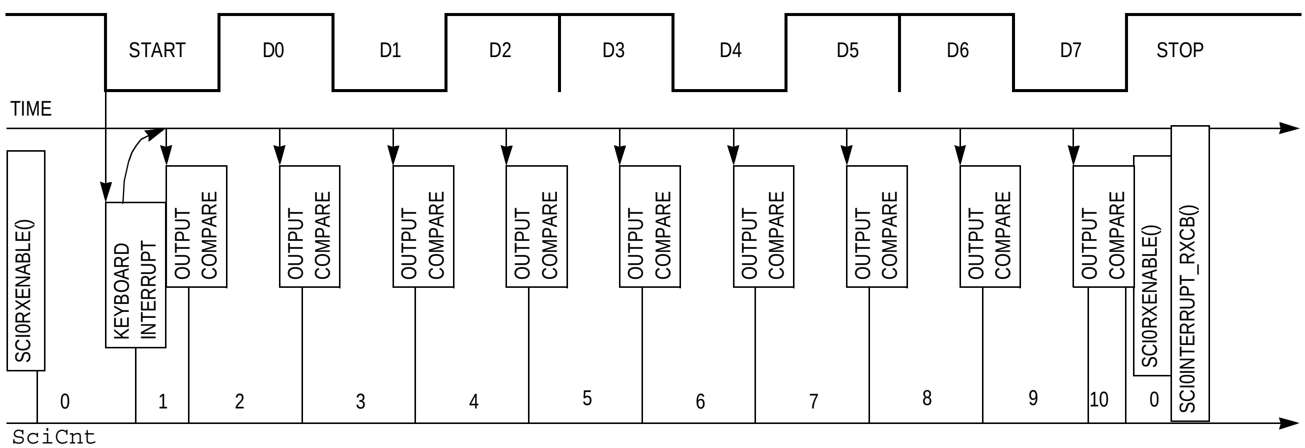


Figure 12. Keyboard Interrupt Start Bit Detection Software Time Chart

Figure 13 contains the flow charts of routines related to this version of software.

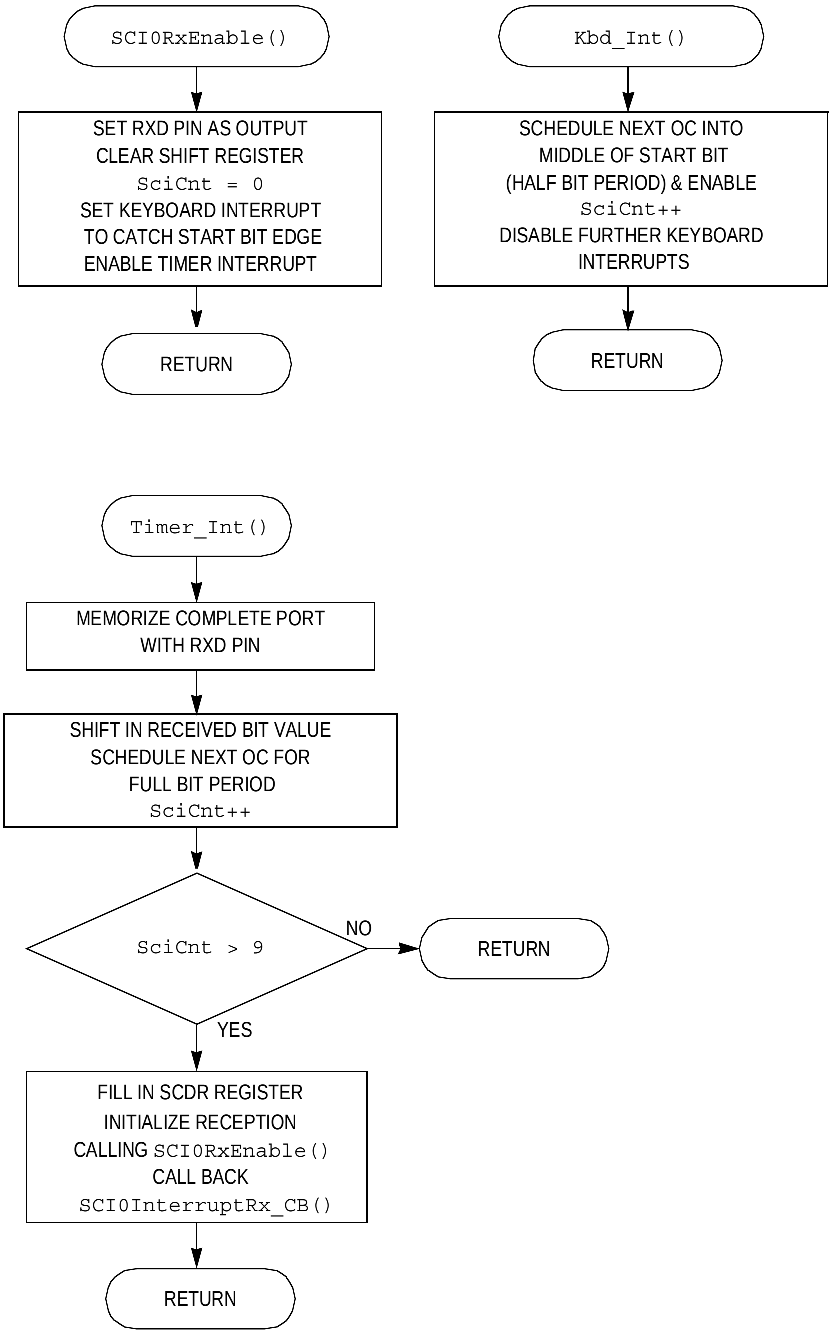


Figure 13. Keyboard Interrupt Start Bit Detection Software Flow Chart

NOTE: *Because the keyboard interrupt hardware can only detect falling edges, only the regular polarity levels can be serviced.*

System Limitations

This software SCI implementation sets several limitations to the user application. In other words, software SCI requires that several conditions be met to work correctly. The limitations namely concern the timer channel that is used for communication.

Modulo Limitation

This implementation has been designed for a user application that shares the 16-bit timer module. The user application's primary mode for the 16-bit timer is PWM generation on the other timer channel. The PWM frequency is constant and relatively low during the application execution.

The software SCI routines do not directly use the modulo feature (overflow feature), but they take into consideration that the user application sets the modulo to some (constant) value. To calculate the timer value for the next interrupt event, the routines must know that modulo value, using `#define` directive:

directive TMRMODULO

Specific Modulo Values

This particular requirement allows a flexible selection of the PWM frequency, so rounded binary values for the modulo counter could be used. Here, the $2^n - 1$ values (such as `0x00FF, 0x01FF, 0x03FF, 0x07FF, 0x0FFF, 0x1FFF, 0x3FFF, 0x7FFF, 0xFFFF`) are effectively implementing modulo calculations by a simple logical AND operation:

```
modulo_value = nonmodulo_value & TMRMODULO;
```

where `nonmodulo_value` could be higher than `TMRMODULO`. This is very effective on M68HC08 arithmetic and requires only two assembly ASM instructions (one of which is removed during the optimizations).

If the user application requires any other value, all modulo calculations must be rewritten into standard modulo C function:

```
modulo_value = nonmodulo_value % TMRMODULO;
```

This implementation then uses the C library modulo function, which is much longer. It will possibly work too, but it has never been tested for the time consumption.

Modulo Can't Be Less...

There is another condition for the modulo value selection. To generate the SCI speed, for example 9600 bps, the PWM frequency must not be higher than 9600 Hz. In this case, the distance between two SCI bits is longer than the total modulo timer cycle. This implementation would not work under such conditions.

In the case of QT/QY M68HC08 running an internal oscillator (at 3.2 MHz bus clock) and 9600 bps baud rate, the limitation for modulo value is 3.200.000/9.600 = 333 (`0x014D`), so the modulo values `0x01FF`, `0x03FF`, `0x07FF`, `0x0FFF`, `0x1FFF`, `0x3FFF`, `0x7FFF`, and `0xFFFF` are feasible.

Timer Must Run All the Time

Another system limitation is that the user software must not stop the timer. In such a case, the software SCI would stop working too.

The user code must also carefully reconfigure the timer registers, so it will not modify any setting that might affect the software SCI timer settings.

Interrupt Latency Low, Interrupts Enabled

Except for output compare driven transmit software, all other versions of the software access serial lines using direct I/O instructions. This means that if the software SCI timer interrupt is delayed for some reason (such as another interrupt being serviced), the SCI signals are delayed too. This delay may lead to corruption of the character being sent/received. The number of other interrupt service routines should be kept to a minimum (which is good practice anyway), or they should be implemented as interruptible.

If the user application must disable interrupts, the amount of off-time should also be kept to a minimum.

No detailed numbers (time restrictions) are provided in this application note because no measurement of error rate was carried out.

In the case of PC master communication that runs over this software SCI, the protocol is tolerant to SCI errors, and sporadic errors are corrected by repeating the data transmission.

Shared Keyboard Interrupt

If the user application also uses the keyboard interrupt, some adjustments must be implemented to share one interrupt service routine between the software SCI and the user code.

System Implementation Notes

This chapter describes some specific system implementation notes.

Internal Oscillator Usage

This application uses the internal oscillator of the QT/QY M68HC08 MCU. The internal oscillator is specified to run at 12.8 MHz, ±25%. The bus clock is then 3.2 MHz, ±25%.

The ±25% variation can be reduced to ±5% by trimming the oscillator. The MCU has a factory pre-programmed trim value at address `0xFFC0` that has been measured at the time of testing. However, there is no guarantee that this value will work with SCI communication.

Another option is to use the developer's serial bootloader (as described in AN2295/D: *Developer's Serial Bootloader*). During bootloading, the correct SCI timing constant is measured and stored in FLASH memory.

The correct SCI timing constant can be retrieved after the bootloader's `SCISPIInit()` routine is called. (Bootloader `sci.h` header file must be included in the project.) The initialization routine will pre-load several variables in RAM, including `SCIAPISpeed`. This is later copied to the internal `BAUDTICK` variable as shown in the following example:

```
#ifdef BOOTLOADERSCIAPIUSED
    SCIAPIInit();           // initialize SCI API
    BAUDTICK = SCIAPISpeed; // get SCI calibrated value (best known)
#endif
```

This process provides a very reliable SCI timing value, based on previous MCU communication with PC, that has a precise and known data rate.

References

AN1948/D: *Real Time Development of MC Applications using the PC Master Software Visualization Tool*

AN2263/D: *PC Master Software: Creation of Advanced Control Pages*

AN2395/D: *PC Master Software Usage*

AN2471/D: *PC Master Software Communication Protocol Specification*

AN2295/D: *Developer's Serial Bootloader for M68HC08*

Source Code Listings

pcmastersoftsci.h:

```
/*************************************************************************
*
* Freescale Semiconductor, Inc.* (c) Copyright 2003 Freescale Semiconduct,  Inc.
* ALL RIGHTS RESERVED.
*
**************************************************************************
*
* $File Name: pcmastersoftsci.h$
*
* Description: Software SCI headers for
*              PC Master Communication protocol
*
* $Version: 1.1.3.0$
* $Date: Sep-25-2003$
* $Last Modified By: r30323$
*
*************************************************************************/
#include "map.h"

/* Software SCI API */
extern char SCI0Read(void);
extern void SCI0Write(char ch);
extern void SCI0RxEnable(void);
extern void SCI0InterruptTx_CB (void);
extern void SCI0InterruptRx_CB (void);
extern voidSCI0Init(void);
/* Software SCI API end */

#pragma DATA_SEG SHORT _DATA_ZEROPAGE
/*------SCI definitions registers-------------------*/
extern char SCDR; /*SCI data register*/
#pragma DATA_SEG DEFAULT

#define BUS_CLOCK_HZ 3200000 /* reqd' bus clock in Hz */

/*###################################*/
/*###################################*/

/*###################################*/
#define BAUDRATE 9600L

#define TMRMODULO 0x3fff        // specify the modulo (mask) in which 'free' running timer
operates
/*###################################*/

/*###################################*/
/*### common softSCI section */
#define SCISINGLEWIRE           // define only if RXD & TXD pins are shared (ie. single wire)
#define SCIINV                // define this one, if SCI needs to be inverted (ie. non-standard
interface)
/*###################################*/
```

```
/*##################################*/
/*### TXD pin section */
#define SCITXDPINISTIMERPIN     // defined if TXD pin can use hw output compare feature
/*##################################*/

#ifndef SCITXDPINISTIMERPIN
#define TXDPIN      PTA0
#define TXDPINDDR   DDRA_BIT0

#define TXDPINPUE   PTAPUE_BIT0
#endif

/*##################################*/
/*### RXD pin section */
#define SCIRXDPINISTIMERPIN     // defined if RXD pin can use hw input capture feature
#define RXDPIN      PTA0
#define RXDPINDDR   DDRA_BIT0
#define RXDPINPORT  PTA
#define RXDPINMASK  0x01
/*##################################*/

#ifndef SCIRXDPINISTIMERPIN  // if RXD is not timer pin, it must be KBI pin
    #ifdef SCIINV
        #error "Cannot use SCIINV and !SCIRXDPINISTIMERPIN features together!"
    #endif
    #define KBIECH KBIER_KBIE0  // and you must define your KBIE here
#endif

/* softSCI timer selection section */
/* must be one of timer channels, if SCIRXDPINISTIMERPIN and/or SCITXDPINISTIMERPIN
   macros are defined, it must also match the appropriate hardware (pin) */
#define SCITSC      TSC
#define SCITSCCH    TSC0
#define SCITSC_CHF  TSC0_CH0F
#define SCITSC_IE   TSC0_CH0IE
#define SCITCNT     TCNT
#define SCITCH      TCH0
#define SCITMOD     TMOD
#define IV_SCITMR   IV_TCHO
/* end */

#define DDRIN 0
#define DDROUT 1

#ifndef SCIINV
    #define TXDPINSET() TXDPIN=1
    #define TXDPINCLR() TXDPIN=0
#else
    #define TXDPINSET() TXDPIN=0
    #define TXDPINCLR() TXDPIN=1
#endif

#define SCITX 1
#define SCIRX 2
```

pcmastersoftsci.c:

```
/*****************************************************************************
*
* Freescale Semiconductor, Inc.
* (c) Copyright 2003 Freescale Semiconductor, Inc.
* ALL RIGHTS RESERVED.
*
******************************************************************************
*
* $File Name: pcmastersoftsci.c$
*
* Description: Software SCI library for
*              PC Master Communication protocol
*
* $Version: 1.1.5.0$
* $Date: Oct-21-2003$
* $Last Modified By: r30323$
*
*****************************************************************************/
#include "map.h"
#include "pcmastersoftsci.h"

#include "pcmaster.h"
#include "pcmasterconfig.h"

#define BOOTLOADERSCIAPIUSED
/* if you undefine this you have to ensure that SCI will get the correct ticks for SCI speed ;-)
   if defined, MCU must be bootloader enabled (Freescale AppNote AN2295) and it will provide
   the proper SCI constant derived out of bootloading communication .... */

#ifdef BOOTLOADERSCIAPIUSED
#include "sci.h"   /* Bootloader's API needed! */
#endif

#pragma DATA_SEG SHORT _DATA_ZEROPAGE
/*------SCI definitions registers-------------------*/
char SCDR; /*SCI data register*/

unsigned char SciBuff;
unsigned char SciPort;
unsigned char SciCnt;
unsigned char SciStat;
unsigned int SciTmr, BAUDTICK;
#pragma DATA_SEG DEFAULT


void SCI0Init(void)
{
#ifdef BOOTLOADERSCIAPIUSED
    SCIAPIInit();           // initialize SCI API
    BAUDTICK = SCIAPISpeed; // get SCI calibrated value (best known)
#else
        BAUDTICK = BUS_CLOCK_HZ / BAUDRATE;
#endif
```

```
    SCITMOD = TMRMODULO;
    SCITSC = 0;             // run timer, no prescaling, no modulo int.
    SCI0RxEnable();

#ifndef SCITXDPINISTIMERPIN
    TXDPINPUE = 1;           // enable pull-up
#endif
};

char SCI0Read(void)
{
    return SCDR;
}

void SCI0Write(char ch)
{

#ifndef SCIRXDPINISTIMERPIN
    KBIECH = 0;              // disable RX KBI int'
#endif
  #ifndef SCIINV
    SCITSCCH = 0x00;         // reset timer logic so no false edge appears
  #else
    SCITSCCH = 0x10;         // reset timer logic so no false edge appears
  #endif
#ifndef SCITXDPINISTIMERPIN
    TXDPINSET();             // just make sure no glitch (high to low) appears
    TXDPINDDR = DDROUT;      // TXD pin output
#endif
    SciStat = SCITX;
    SciBuff = ch;            // copydown the timer
    SciCnt = 9;              // 8 bits of data + stop bits to send

#ifdef SCITXDPINISTIMERPIN
  #ifndef SCIINV
    SCITSCCH = 0x18;         // output compare, falling edge
  #else
    SCITSCCH = 0x1C;         // output compare, rising edge
  #endif
    SCITCH = SciTmr = ((SciTmr = SCITCNT) + BAUDTICK) & TMRMODULO;
#else
    SCITSCCH = 0x10;         // just timer int to be scheduled (just port control)
    SCITCH = SciTmr = ((SciTmr = SCITCNT) + BAUDTICK) & TMRMODULO;
    TXDPINCLR();             // TXD pin low (start bit)
#endif

    SCITSC_CHF = 0;         // clearing timer flag
    SCITSC_IE = 1;           // enable tmr. channel interrupts
}

void SCI0RxEnable(void)
{
    SCITSC_IE = 0;           // disable tmr. channel interrupts

    RXDPINDDR = DDRIN;       // RXD pin input
```

```
    SciStat = SCIRX;
    SciCnt = 0;              // sci cnt will be falling edge

#ifdef SCIRXDPINISTIMERPIN
  #ifndef SCIINV
    SCITSCCH = 0x08;         // input capture, falling edge only on tmr.
  #else
    SCITSCCH = 0x04;         // input capture, rising edge only on tmr.
  #endif
    SCITSC_CHF = 0;         // clearing timer flag
    SCITSC_IE = 1;           // enable tmr. channel interrupts
#else
    /* specify RXD fallling edge interrupt init here! */
    KBSCR_IMASKK = 1;        // mask int now (safe int init)
    KBSCR_MODEK = 0;         // edge only
    KBIECH = 1;              // enable pin specific KBI int'
    KBSCR_ACKK = 1;          // confirm interrupt
    KBSCR_IMASKK = 0;        // unmask int now
#endif
}

#ifndef SCIRXDPINISTIMERPIN
void interrupt IV_KBRD Kbd_int(void)
{
    SCITCH = SciTmr = ((SciTmr = SCITCNT) + BAUDTICK/2) & TMRMODULO;

    SciCnt++;
    SCITSC_CHF = 0;         // clearing timer flag
    SCITSCCH = 0x50;         // timer int to be scheduled (keep int enabled)

    KBIECH = 0;              // and disable KBI int - all subsequent ints are timer driven
    KBSCR_ACKK = 1;          // confirm interrupt
}
#endif

void interrupt IV_SCITMR Timer_int(void)
{
    SciPort = RXDPINPORT;        // as fast as possible port scan for receive branch

    if (SciStat == SCITX)
    {

#ifdef SCITXDPINISTIMERPIN
        if (SciCnt > 1)
        {
            SCITCH = SciTmr = (SciTmr + BAUDTICK) & TMRMODULO;
            SciCnt--;                  // decrement counter
    #ifndef SCIINV
           SCITSCCH = 0x58 | (SciBuff & 0x01?0x04:0);        // output compare, schedule clear
output (start bit)
    #else
            SCITSCCH = 0x58 | (!(SciBuff & 0x01)?0x04:0);        // output compare, schedule
clear output (start bit)
    #endif
```

```
            SciBuff >>= 1;                // shift internal buffer
        }
        else if (SciCnt == 1)             // stop bit reached
        {
    #ifndef SCIINV
          SCITSCCH = 0x58 | 0x04;     // output compare, schedule set output (stop bit)
    #else
            SCITSCCH = 0x58;     // output compare, schedule set output (stop bit)
    #endif
            SCITCH = SciTmr = (SciTmr + BAUDTICK) & TMRMODULO;
            SciCnt--;                     // decrement counter
        }
        else
        {
            SCITSCCH = 0x00;         // port control, set output & disable further interrupts
            SCI0InterruptTx_CB();
        }
#else /* ifdef SCITXDPINISTIMERPIN */
        if (SciCnt > 1)
        {
    #ifndef SCIINV
            TXDPIN = SciBuff & 0x01;    // copy to TXD pin
    #else
            TXDPIN = ~(SciBuff & 0x01);    // copy to TXD pin
    #endif
            SciBuff >>= 1;                // shift internal buffer
            SCITCH = SciTmr = (SciTmr + BAUDTICK) & TMRMODULO;
            SciCnt--;                     // decrement counter
        }
        else if (SciCnt == 1)    // stop bit reached
        {
            TXDPINSET();         //stop bit
            SCITCH = SciTmr = (SciTmr + BAUDTICK) & TMRMODULO;
            SciCnt--;                     // decrement counter
        }
        else
        {
    #ifdef SCISINGLEWIRE
            TXDPINDDR = DDRIN;     // TXD pin input
    #endif
            SCITSC_IE = 0;           // disable further interrupts
            SCI0InterruptTx_CB();
        }
#endif /* ifdef SCITXDPINISTIMERPIN */

    }
    else /* if (SciStat == SCITX) */
    {

#ifdef SCIRXDPINISTIMERPIN
        if (SciCnt == 0)         // start bit falling edge captured
        {
            SCITSCCH = 0x50;    // timer int to be scheduled (keep int enabled)
            SCITCH = SciTmr = ((SciTmr = SCITCH) + BAUDTICK/2) & TMRMODULO;
            SciCnt++;           // first int will be useless (in the middle of start bit)
```

```
        }
        else
#endif /* ifdef SCIRXDPINISTIMERPIN */
        {
    #ifndef SCIINV
                if (SciPort & RXDPINMASK)
    #else
                if (!(SciPort & RXDPINMASK))
    #endif
                SciBuff = (SciBuff>>1) | 0x80;
            else
                SciBuff = (SciBuff>>1) & 0x7f;
            SCITCH = SciTmr = (SciTmr + BAUDTICK) & TMRMODULO;
            SciCnt++;

            if (SciCnt > 9)     // 9 bits because first is in start bit (*not used*)
            {
                SCDR = SciBuff;         // copy down the received buffer
                SCI0RxEnable();         // restore RX
                SCI0InterruptRx_CB();       // make RX interrupt!
            }
        }
    }  /* if (SciStat == SCITX) */

    SCITSC_CHF = 0;         // clearing timer flag
}
```

[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