Order this document
by AN2025/D

Motorola Semiconductor Application Note

AN2025

A General-Purpose Serial Communication Interface for the MMC2001 Using the Motorola API

By Hans-Guenter Kremser
WSSG
Munich, Germany

Introduction

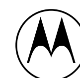
Even though many microcontrollers (MCU) have an integrated UART (universal asynchronous receiver transmitter) on board, some customers may need another one. For instance, if the part has only one UART, they may need two.

One way to overcome this problem is to design a software UART, also called SCI (serial communication interface). This is feasible if the MCU has a general-purpose timer like the MMC2001 MCU does, with its six independent PWMs (pulse-width modulator).

This application note describes a software (SW) UART, written in C language. It has been developed for the MMC2001, a Motorola M•CORE MCU. Because it is a program written in a high-level language, the SW UART can be adapted easily to other cores like PowerPC® and 68HCxx.

This SCI is designed for an 8-bit data, one start bit, one stop bit, least significant bit (LSB) first protocol, which is probably the most common

PowerPC is a registered trademark of International Business Machines Corporation.



MOTOROLA

© Motorola, Inc., 2000

configuration. The baud rate (bits per second) can vary up to a maximum of 19,200 baud. The protocol needs to be set before establishing a communication link. The SCI can communicate only in half-duplex mode. This means it can either transmit or receive, but it cannot do both functions simultaneously.

For more information, several Motorola application notes describe the RS-232 protocol, including "HC05 MCU Software-Driven Asynchronous Serial Communication Techniques Using MC68HC705J1A," document order number AN1240/D; "Software SCI Implementation to the MISC Communication Protocol," number AN1667/D; and "Software SCI Routines with the 16-Bit Timer Module," order number AN1818/D.

Motorola Application Programming Interface (API)

Many development tools for the M•CORE are available on Motorola’s Web site at www.mcu.motsps.com. Select Documentation. Also, there is a link to the device driver library, which is used to program the described SCI.

Using the device driver library reduces development time and provides:

- Documentation with example programs for each function. Because all documentation is available electronically, cut and paste can be used.
- Each peripheral has its own “include” file. All addresses and structures of the registers are defined in these files. Sometimes it makes sense to use these structures directly without using the library function.
- The library file containing the functions (a file called libplib.a). Using CodeWarrior® makes it easy to include this library. Click the run time folder in the project folder and select “add file…” under the project button in the menu bar.

CodeWarrior is a registered trademark of Metrowerks, Inc., a wholly owned subsidiary of Motorola, Inc.

- C source code examples for each peripheral with the appropriate batch file for compiling, linking under a Diab Data/SDS (now WindRiver) or GNU environment. Also, a tutorial program is included.

Another advantage of using this API is that the address of each register and subregister doesn't need to be known. Even the program status register (PSR) could be accessed within the C source code by using the appropriate function. Because each function returns a value that indicates successful completion, error handling can be used.

In an effort to ensure the functions used in this software are better understood, this application note describes how the API sets each bit on the register level.

These device drivers also are available for the MMC2003 (includes same functions as the MMC2001, as well as a correlator for GPS (global positioning system) functionality), the MMC2080 (MCU for pagers supporting the FLEX™ protocol), the MMC2107, and future M•CORE-based MCUs.

The Concept

During transmission and reception, every bit is sent or received via an interrupt. The advantage of this concept is quite obvious: no CPU allocation.

If one transmits at 9600 baud without using interrupts, the CPU would be allocated for about 1 ms per character (1/9600 times 10 bits) only for this task.

While using the interrupt approach, the CPU is free for other tasks. The disadvantages of this technique is a little jitter between the bits while transmitting, and, during reception the sampling time may differ from bit to bit. But these are mostly meaningless as long as one stays in the specified drift range. Interrupt priorities for the different tasks in the system need to be assigned in the application program.

FLEX is a trademark of Motorola, Inc.

Because the MMC2001 is based on the M•CORE, a 32-bit RISC (reduced instruction set computer) core, the interrupt handling is different compared to other MCUs, like those in the M68HC05 Family, for instance.

This application note uses the interrupt autovector. Each time any interrupt occurs, this autovector is called. The program is responsible for determining the interrupt source. In the described SCI, this was done by using the “Set Interrupt Service Routine” functions provided by the library.

If the user prefers to program in assembly, an easy way to find the interrupt source is the “Find First One” command (FF1 ISR) and apply it to the interrupt source register (ISR). It returns the offset of the bit position beginning at the most significant bit (MSB). But because the Motorola library is used, not a single line is programmed in assembly.

In general, several steps are necessary to generate an interrupt. They are:

1. Set the exception enable bit (EE) in the processor status register (PSR).
2. Set either the interrupt enable bit (IE) or the fast interrupt enable bit (FE), or both, in the PSR.
3. Set the appropriate bit for the peripheral where generation of an interrupt is desired (for instance, PWM5, bit number 15) in the normal interrupt enable register (NIER), or fast interrupt enable register (FIER), or in both.
4. Finally, the interrupt has to be enabled in the peripheral’s control register (for instance, bit number 9 in the PWMCR5).

PWM channel 4 is used for receiving and channel 5 is used for transmitting. Both PWMs are used only to maintain the timing and generate an interrupt.

The associated hardware pins are not used in PWM mode. The transmitter uses the PWM5 output pin, which is used as a general-purpose input/output (/O). The INT6 pin of the EDGE port is used for the receiver.

Implementation in Brief

To transmit data, a function is called. This function enables the interrupt for PWM5 and starts the timer. Each time the period of PWM5 reaches its value, an interrupt is generated. This interrupt calls a service routine which transmits the data ready for transmission bit by bit until the last bit of the last data word has been sent.

To receive data, a valid start bit has to be detected at the edge port pin. This means a falling edge and a logical 0 in the middle of the start bit. As soon as this is confirmed, PWM4 is activated and the timer starts to check for the values in the middle of the next eight data bits.

Figure 1 shows how the main program is organized and is an example of a possible implementation. Notice that no receive function is called because it is unknown when data arrives. The main program is ready for reception during the endless loop.

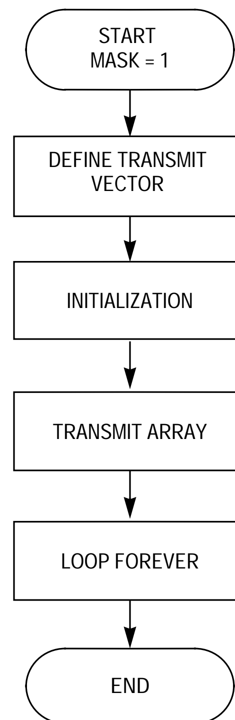


Figure 1. Main Program Implementation

Initialization

During the initialization, the following things have to be done:

1. Transmit PWM control register — Setting up the division ratio, which is the ratio between processor clock frequency and the PWM clock. It can be adjusted by three bits, so $2^3 = 8$ coarse ratios are possible. Enable the interrupt bit and configure the port pin as a general-purpose output pin. The MMC2001 provides six independent PWMs (for instance, each PWM can be programmed with different frequency and duty cycle). The transmitter makes use of PWM5, which starts at address 0x10005028 (0x indicates hexadecimal) with its control register. All pointers are set in global assignments, right after the #include section.

Table 1 describes the configuration of the PWM control register 5 (PWMCR5).

Table 1. PWM Control Register 5 Setup

Bit	Description	Setting
14–12	Reserved	0
11: Doze mode (DOZE)	Bit = 1: Disabling this PWM channel in doze mode	0
10: PWM interrupt request (PWMIRQ)	Bit = 1: Forcing an interrupt, for instance, for debugging	0
9: Interrupt request enable (IRQEN)	Bit = 1: An interrupt is generated each time the period is completed	1
8: Load period and width register (LOAD)	Bit = 1: Forcing a new period immediately	0
7: PWM data (DATA)	If this PWM channel is configured as a general-purpose I/O (GPIO), the pin value is reflected by this bit.	1
6: Direction (DIR)	If used as GPIO, a 1 sets the pin to output, a 0 to input.	1
5: Polarity (POL)	Only valid when used in PWM mode; 0 = normal polarity; 1 = inverted polarity	0
4: Mode (MODE)	0 = general-purpose I/O mode; 1 = PWM mode	0
3: Counter enable (COUNTEN)	Setting this bit starts the timer.	0

Table 1. PWM Control Register 5 Setup (Continued)

Bit	Description	Setting
2–0: Clock select (CLKSEL)	000 sets dividing ratio to 4 (32-MHz system clock / 4 = 8 MHz). 8 MHz is the reference clock for generating the baud rate. If the SCI has to be set for a data rate of 9600 baud, calculate 8 MHz / 9600 = 833.33 to get the right value to program the period register. Other configurations are possible: division ratio of 8 (001) results in 4 MHz f_{ref}. 4 MHz / 9600 = 416.66 The only thing to consider is that the timing resolution decreases by increasing the pre-divider ratio.	000

The actual setting of this register is done by using the API function PWM_A_SetRegister (), which has three parameters:

a. PWM base address

b. Actual PWM register; each PWM has four 16-bit registers which are control, period, width, and timer register

c. Data, which could be 32 bits wide, making it possible to write to two registers in one write statement

2. Transmit PWM period and width registers — These are 10-bit registers (upper six bits are read only) that fine tune the PWM. Because only the period is needed to generate an interrupt, the value in the width register doesn't matter as long as it is smaller than the period value. As already discussed, a period value of 833 is selected.

   Also, the calculation can be done in a different way: A reference clock of 8 MHz is equal to a period of 125 ns. For example, it takes 125 ns each time the counter is increased by one step. At a baud rate of 9600, each bit needs 104.16 µs and 104.16 µs ÷ 125 ns = 833. Transmitting 10 bits (one start, eight data, and one stop bit) takes 1.04 ms.

The two registers are set with the PWM_A_UpdateOutput() function. With this function, it is possible to change the LOAD bit in the PWM control register and write to the period and width registers.

3. Setting the TX pin to high, which is the idle condition — This is realized with a macro in C because the transmit pin must be set and cleared often. Two macros are defined: SCI_TX_ON, which sets the pin to logical high, and SCI_TX_OFF, which clears the pin.

   The definitions for these two macros are done in the main.h include file. In this case, they are directly programmed in C without using any API function, but using the API acronyms like PWMCR and PWM_A_DATA_MASK. To switch the pin to high bit number 7 in the PWM5 control register is “binary OR’ed” with the PWM_A_DATA_MASK. It is cleared by “logical ANDing” the register with the inverted PWM_A_DATA_MASK.

4. Receive PWM control register — Setting up the division ratio (the same as in transmit PWM) and enabling the interrupt if the counter reaches the period value. No other bit has to be set in this register. The associated pin is set to GPIO and input by default. This pin will not be used because the external interrupt will be used as the receiver input pin. PWM channel 4 is used for the receiver. The period is set to half the value of the transmit period because the user needs to sample in the middle of a possible start bit. After this, the period needs to be changed again to the transmit value.

5. Setting the vector base address — The M•CORE supports many different interrupt sources, which have dedicated offsets. These offsets have to be added to the base address. The API function INTC_A_Init () sets the interrupt vector base address to the start of the internal RAM address, which is 0x30000000. The offset is 0x2C because the fast interrupt autovector is used. So, each time any fast interrupt occurs, the program fetches the address at 0x3000002C and then vectors to that address.

6. The interrupt handling is assigned in a library function called INTC_A_SetISF, which checks if the interrupt source bit is flagged and assigns a dedicated interrupt service routine name. This function name is called when the interrupt occurs.

Three interrupt service routines have to be assigned:

a. sci_tx() for the transmitting PWM — Will be called when the transmit PWM5 generates an interrupt

b. sci_rx() for the receive pin — Will be called when an external interrupt occurs at the INT6 pin

c. sci_receive() for the receiving PWM – Will be called when the receive PWM4 generates an interrupt

7. Finally, the interrupts have to be enabled. This is also done with the library function INTC_A_IntEnable(). It sets the above mentioned three bits (PWM4, PWM5, and INT6) in the fast interrupt enable register (FIER) and enables the EE and FE bits in the processor's program status register (PSR).

8. The external interrupt pin, INT6, needs to be configured as an input pin and as falling edge sensitive because this could be the start of a valid RS-232 protocol. In this case, no API function is called. Instead, it is realized manually by following the edge port pointer to the edge port pin assignment register (EPPAR) and masking it with the appropriate mask, which for the INT6 external pin is EPPAR_EPPA6_FALLING_EDGE_MASK.

Transmitter

The transmitting part is split into two functions. They are:

1. The function that set the transmit array and its length

2. The function that puts the received data into a global array and does the interrupt-driven transmission

The real transmission is handled by the function sci_tx, which is called each time the PWM5 matches the value in its period register. This will be the case every 104 µs. Some functions in Motorola's device driver library allow passing variables to interrupt functions. This could have been used, but using global variables in this application note solves the problem of passing variables to interrupt routines. A state machine (a strong word for a counting variable) was implemented to differentiate

among the start bit, eight data bits, and stop bit. The transmission stops by stopping the associated PWM as soon as the last bit (MSB) of the last byte in the transmit array is reached.

The flowchart in **Figure 2** shows an overview of the transmitter (tx is used for transmitter; rx is used for receiver).

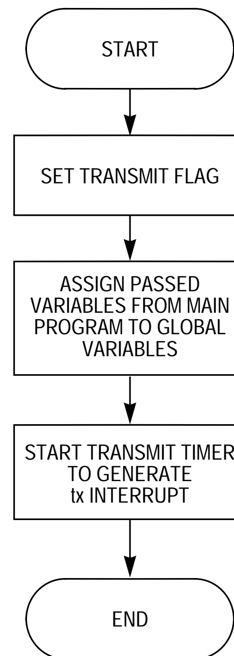


Figure 2. Transmit Function Used in the Main Program

The transmit function calls the bit transmit function. The flowchart for the bit transmit function is shown in **Figure 3**.

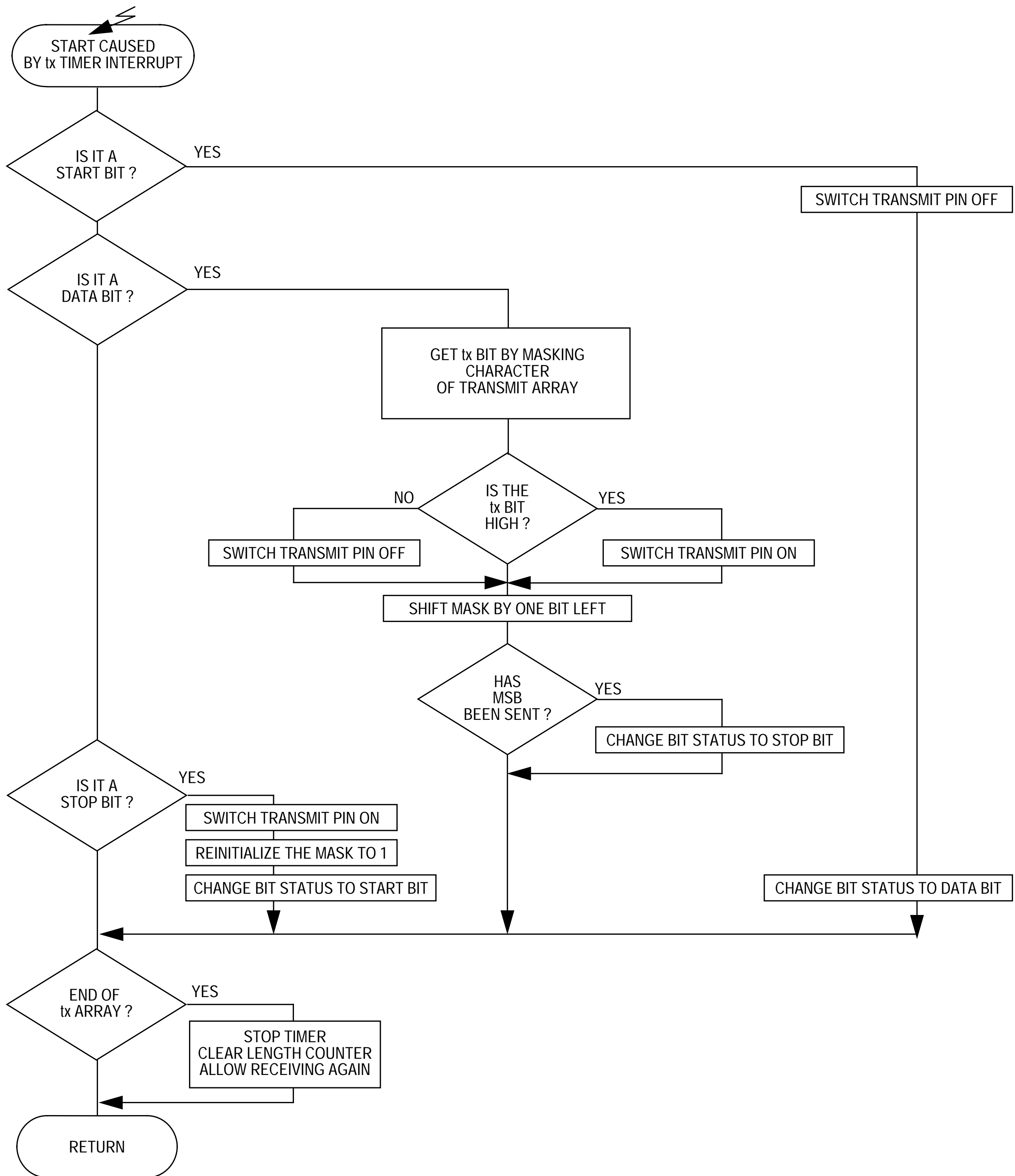


Figure 3. Bit Transmit Function, Called by Transmit Timer Interrupt

Receiver

The receiver is slightly more difficult, since the time that a message will be received is unknown. The best solution is to use an external interrupt pin.

We will use the fact that a received message always starts with a negative transition followed by logic 0 (start bit) and configure the interrupt pin for negative transition detection.

But what if it was just a spike? This problem is solved by a timer which starts at the negative slope and looks in the middle of the start bit to see if it is still logic 0. It is unlikely another spike will occur exactly at this time. At this point, a flag indicating valid data reception could be set. Now the timer has to be changed to the bit period to look at the data value in the middle of each bit. Another PWM (PWM4) interrupt is used to look for the eight bits of data. The receiving buffer is filled as long as valid data is detected at the external interrupt pin.

Figure 4 shows how the receiver is realized.

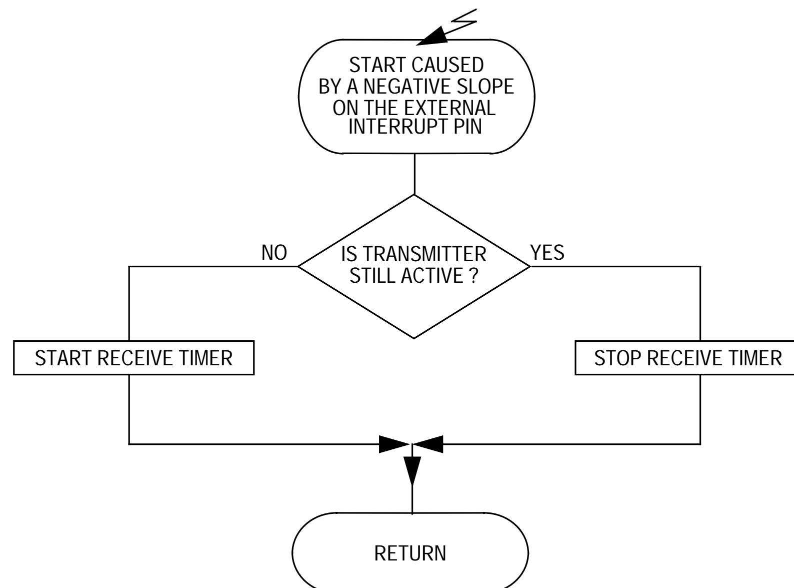


Figure 4. Receive Function, Called by External Interrupt

The receive function calls the bit reception function, shown in **Figure 5**.

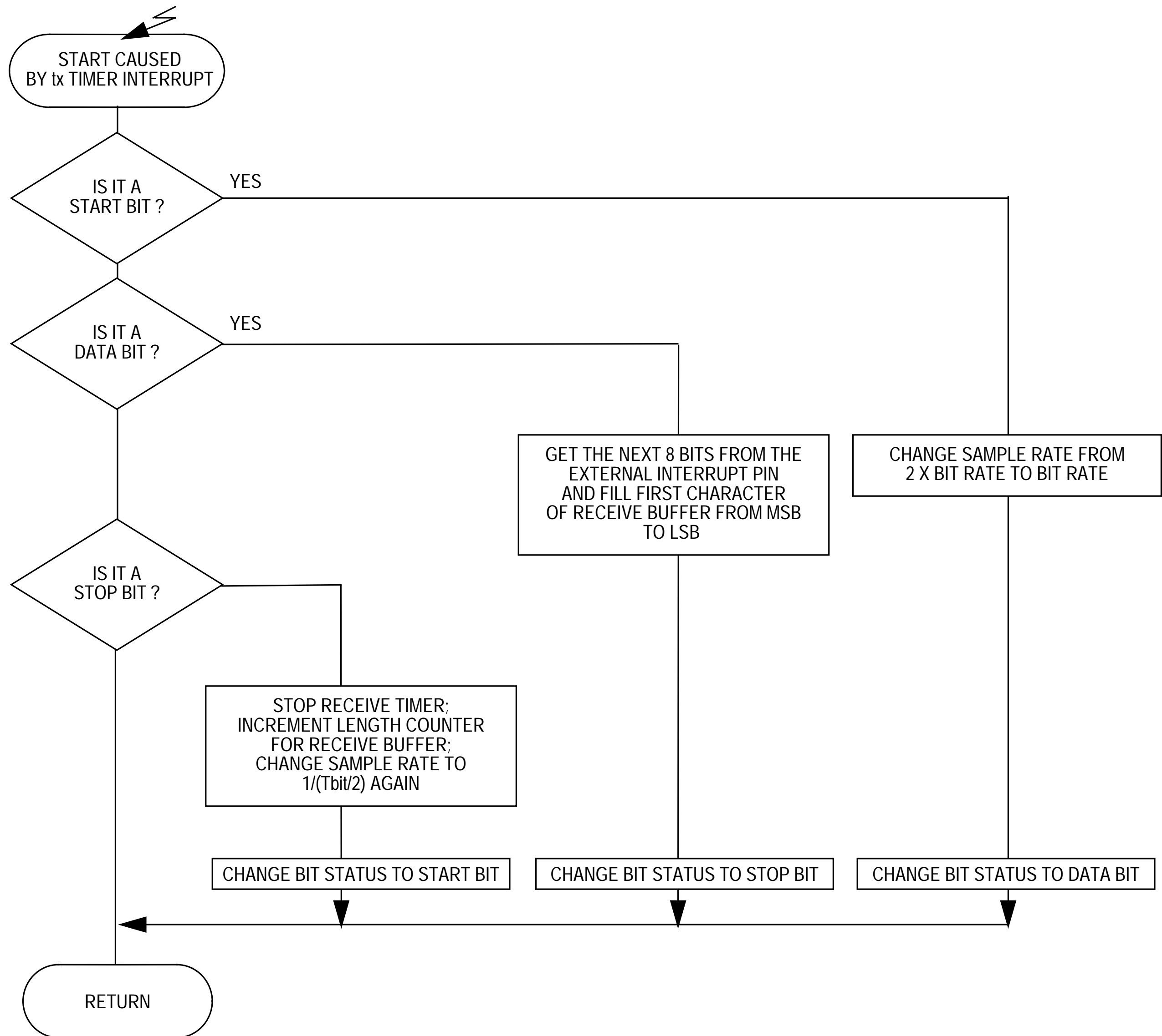


Figure 5. Bit Reception Function, Called by Receive Timer Interrupt

Summary

This application note describes how to realize a half-duplex software UART, completely written in the high-level language C. It uses the Motorola device driver library which is accessible on the Motorola Web site. All bits are transmitted and received by interrupt functions. This application note demonstrates a good concept for implementing these routines in an operating system or callable in an application.

All the functions and the main program are located in the file main.c, and the main.h file contains local definitions. All other header files which the main.c file includes are provided by the device driver library.

The linker command file called linker.cmd handles the memory organization. In this file, the RAM, ROM, vectors, and stack addresses are defined. CodeWarrior from Metrowerks provides the file.

The code has been tested with a Motorola MMC2001 evaluation board (MMCEVB1200PV) and the Motorola EBDI (enhanced background debug interface) in a Metrowerks CodeWarrior environment.

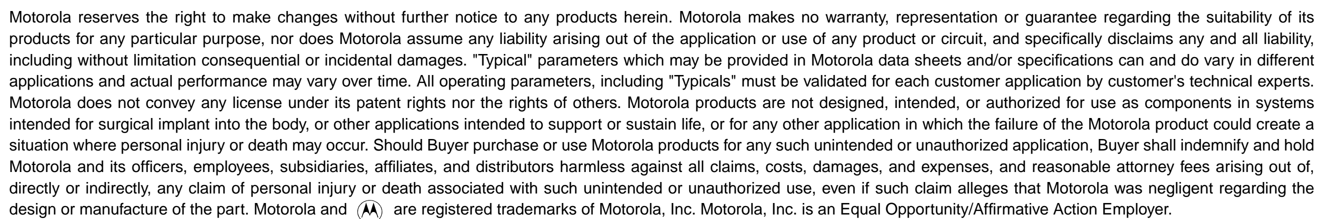
Source Code

```
/*
*  Serial Communication Interface
*  filename: main.c
*  PWM5 with PWM5 pin used for TX
*  PWM4 with INT6 pin used for RX
*/

#include "plibdefs.h"
#include "plib.h"
#include "pwm_a.h"
#include "intc_a.h"
#include "edgeport_a.h"
#include "main.h"


// global initializations (provided by the API)
pPWM_A_t tx_pwmptr = (pPWM_A_t) __PWS_PWM5;
pPWM_A_t rx_pwmptr = (pPWM_A_t) __PWS_PWM4;
static intTbl_t funcs;                          // interrupt and signal service tables
pINTC_A_t intctlr = (pINTC_A_t) __PWS_INTCTLR;
pEdgePort_A_t edgeportptr = (pEdgePort_A_t) __PWS_EdgePort;


/*
Function name: sci_tx
Purpose: this function is called every time the PWM5 timer generates an interrupt (every 104us)
         it transmits the data in the global array tx_buffer[] bit by bit.
Input:   global array tx_buffer
Output:  PWM5 GPIO is used as the transmit pin
*/
void sci_tx (void) {
        tx_pwmptr->PWMCR & PWM_A_IRQ_MASK;        // clear PWM interrupt request bit
        switch (tx_state) {
                case 0:
                         SCI_TX_OFF;              // start bit
                         tx_state=1;
                         break;
                case 1:
                         if (tx_buffer[j] & mask)// LSB first
                                SCI_TX_ON;
                         else
                                SCI_TX_OFF;
                         mask <<= 1;
                         if(!mask)
                                tx_state=2;       // last data bit (MSB) been sent
                         break;
                case 2:
                         SCI_TX_ON;               // stop bit
                         mask=MASK_INIT;
                         tx_state=0; j++;
        }
        if (j == tx_length) {                     // if end of array, stop transmission
                PWM_A_Stop (tx_pwmptr);
                j=0;                              //clear length counter for next message
                transmit=FALSE;                   // allow receiving again
        }
}
```

```
/*
Function name: sci_rx
Purpose: this function is called by an interrupt on INT6 pin. It allows reception only while
         not transmitting
Input:   none
Output:  none
*/
void sci_rx(void) {                                  // function called by first falling edge on
         INT6 pin
        edgeportptr->EPFR |= EPFR_EPF6_MASK;         // clear flag!!
        if (!transmit)
                PWM_A_Start (rx_pwmptr);             // half duplex!!
        else
                PWM_A_Stop (rx_pwmptr);
}

/*
Function name: sci_receive
Purpose: this function is called every time the PWM4 timer generates an interrupt (every 104us)
         and if a valid falling edge has been detected at INT6 pin . The rx_buffer[] is filled
         bit by bit.
Input:   external INT6 pin
Output:  global array rx_buffer
*/
void sci_receive (void) {
        intctlr->FIER &= ~INTSRC_PWM5_MASK;          // disable tx interrupts
        rx_pwmptr->PWMCR & PWM_A_IRQ_MASK;           // clear PWM interrupt request bit
        if ((edgeportptr->EPDR & EPDR_EPD6_MASK) == 0) // sample in the middle of start bit
                rx_pwmptr->PWMPR = 833;              // valid start bit;change sample rate to 9600
                                                     // baud
        if (k==0)
                k++;                                 //skip start bit
        else if (k<9) {
                rx_buffer[rx_length] >>=1;           // first time nothing happens, then shift MS
                                                     // to LSB
                rx_buffer[rx_length] |= (edgeportptr->EPDR & EPDR_EPD6_MASK)<<1;
                                                     // set MSB if  INT6 is set
                k++;
        }
        else if (edgeportptr->EPDR & EPDR_EPD6_MASK) {
                PWM_A_Stop (rx_pwmptr);              // stop bit received
                k=0;
                rx_length++;
                intctlr->FIER |= INTSRC_INT6_MASK;// enable INT6 interrupts
                rx_pwmptr->PWMPR = 416;              // change sample rate to Tbit/2 again
        }
}


/*
Function name: sci_init
Purpose: this function is called in the main program to setup baudrate for receiver
        and transmitter, to enable the appropriate interrupts, to configure the GPIO pins, etc.
Input:   none
Output:  none
*/
void sci_init(void) {
        void *VBA = __PWS_OnChipRamBase;             //start of internal RAM

        PWM_A_SetRegister (                          // initialize divider to :4
```

```
        tx_pwmptr,
        PWM_A_PWMCR_SWITCH,
        tx_pwmptr->PWMCR = PWM_A_IRQEN_MASK + PWM_A_DATA_MASK + PWM_A_DIR_MASK +
         PWM_A_DIV_4
);
PWM_A_UpdateOutput (
        tx_pwmptr,                      // PWM channel 5
        FALSE,                          // do not force new period
        833,                            // period: 32MHz clock / 4 (clock divider) /
                                         833 = 9604 baud
        416                             // ton/toff doesn't matter as the interrupt
                                         is only dependent on period
);
SCI_TX_ON;                              // idle TX line
PWM_A_SetRegister (                     // initialize divider to :4
        rx_pwmptr,
        PWM_A_PWMCR_SWITCH,
        rx_pwmptr->PWMCR |= PWM_A_IRQEN_MASK
);
PWM_A_UpdateOutput (
        rx_pwmptr,                      // PWM channel 4
        FALSE,                          // force new period
        416,                           // set timer to Tbit/2 for sampling in the
                                       middle of start bit
        208                           // ton/toff doesn't matter as the interrupt is
                                          only dependent on period
);
INTC_A_Init (
        intctlr,
        VBA,                            // Vector Base Address
        &funcs                          // Function Table Addr
);
INTC_A_SetISF (                         // for tx
        intctlr,                        // INTC Device Handle
        INTSRC_PWM5_BITNO,              // PWM5 Interrupt Source
        INTSRC_PWM5_MASK,               // PWM5 Interrupt Mask
        (ddErr_t(*)(void *, void *))sci_tx,// interrupt service function address
        NULL,                           /* ISF Dummy Parameter */
        NULL                            /* ISF Dummy Parameter */
);
INTC_A_SetISF (                         // for rx
        intctlr,
        INTSRC_INT6_BITNO,              /* INT6 Interrupt Source */
        INTSRC_INT6_MASK,               /* INT6 Interrupt Mask */
        (ddErr_t(*)(void *, void *))sci_rx, /* interrupt service function address */
        NULL,                           /* ISF Dummy Parameter */
        NULL                            /* ISF Dummy Parameter */
);
INTC_A_SetISF (                         // for rx
        intctlr,                        /* INTC Device Handle */
        INTSRC_PWM4_BITNO,              /* PWM4 Interrupt Source */
        INTSRC_PWM4_MASK,               /* PWM4 Interrupt Mask */
        (ddErr_t(*)(void *, void *))sci_receive,/*interrupt service function address */
        NULL,                           /* ISF Dummy Parameter */
        NULL                            /* ISF Dummy Parameter */
);
INTC_A_IntEnable (
        intctlr,
        INTSRC_PWM4_MASK + INTSRC_PWM5_MASK + INTSRC_INT6_MASK,
        TRUE,                           /* Fast Interrupt */
        TRUE                            /* Enable in PSR */
```

```
        );
        edgeportptr->EPPAR |= EPPAR_EPPA6_FALLING_EDGE_MASK;// INT6 is falling edge sensitive,
                                                    input
};


/*
Function name: sci_send
Purpose: this function is called from the main program
Input:   data to be send and the length of the transmit array
Output:  global array tx_buffer and global variable tx_length
*/
void sci_send(u1 *buffer_ptr, s2 buffer_length) {
        transmit=TRUE;
        tx_length = buffer_length;              // save buffer length
        tx_buffer = buffer_ptr;                 // fill tx_buffer with transmit data
        PWM_A_Start (tx_pwmptr);
}


// example main program for demonstration
// the receiving function could be tested as follows:
// connect e.g. a PC serial port (e.g. COM1) to the INT6 pin of the MCORE evaluation board
// start the main program and let it run into the endless loop
// send data from the PC port by using copy filename.txt COM1 (where filename.txt contains
// the data in ASCII),
// the global array rx_buffer will contain the data

void main (void) {
        u1 p[]={0x81,0xab};                     // initialize the send-array

        sci_init();                             // call the initialize routine
        sci_send(p,2);                          // call the send function and pass the send-
                                               array
        while (1)                               // endless loop, waiting for receive data
;
}

main.h

// Local defines
#define SCI_TX_OFF  tx_pwmptr->PWMCR &= ~PWM_A_DATA_MASK // DATA = 0:led on
#define SCI_TX_ON   tx_pwmptr->PWMCR |= PWM_A_DATA_MASK  // DATA = 1:led off
#define MASK_INIT   0x01 // LSB first
#define RX_BUFFER_LENGTH  5

// global variables
static u1 *tx_buffer;
static u1 rx_buffer[RX_BUFFER_LENGTH];
static s2 tx_length;                            // length of TX buffer
static s2 rx_length;                            // length of RX buffer
static s2 tx_state=0;                           // state machine in sci_tx
static u1 mask=MASK_INIT;                       // mask for TX byte
static u1 j=0,k=0;                              // bit index in byte: j for sci_buffer, k for
                                                 // RX buffer
bool transmit=FALSE;
```



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 which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. 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.

How to reach us:

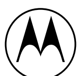
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1-303-675-2140 or 1-800-441-2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu, Minato-ku, Tokyo 106-8573 Japan. 81-3-3440-3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852-26668334

Technical Information Center: 1-800-521-6274

HOME PAGE: http://www.motorola.com/semiconductors/



MOTOROLA

© Motorola, Inc., 2000