



Université d'Ottawa • University of Ottawa

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING

COURSE: CEG3131/CEG3531**SEMESTER:** Fall 2005**PROFESSORS:**

Gilbert Arbez

DATE:

Dec 15, 2005

TIME:

14h00 – 17h00

FINAL EXAMINATION**NAME and STUDENT NUMBER:** _____ / _____**Instructions:**

- Answer ALL questions Note that the exam has been copied on both sides of the page. Spare sheets will be available for draft work. If you need additional space to submit answers, please ask for a booklet.
- This is a close-book examination. **A copy of the CPU 12 Reference Guide is allowed.**
- Use the provided space to answer the following questions. If more space is needed, use the back of the page.
- Show all your calculations to obtain full marks.
- Calculators are allowed.
- Read all the questions carefully before you start.
- Pages starting at page 17 can be detached from the exam.
(total pages 26)

1. There are three (3) types of questions in this examination.

Part 1	Short Answer	20 marks	
Part 2	Theory	20 marks	
Part 3	Application	60 marks	
Total		100 marks	

Part 1 – Short Answer Questions (total 20 points)**QUESTION 1**

Consider the following code. See the Annex A for details on the module.

```

156:  ;----- Buffering Sub-module -----
-
157:  ; Description:
158:  ;      This module provides the means for buffering characters to be
159:  ;      transmitted via the SCI interface. The buffer is found at address
160:  ;      "buffer" with length BUFLen. Two pointers in memory are used to
161:  ;      write (bufEnd) and read (bufStart) characters to/from the buffer.
162:  ;      The variable in memory "numBuf" provides the number of characters
163:  ;      in the buffer that can be transmitted.
164:
165:  ;-----
166:  ; Subroutine: initBuf
167:  ; Global Variables:
168:  ;      bufStart, bufEnd: pointers into the buffer initialized to its
169:  ;      start buffer: address of the start of the buffer
170:  ;      numBuf: number of characters in the buffer
171:  ; Description:
172:  ;      Initializes the pointers to the start of the buffer and numBuf to 0.
173:  ;      This state of the variables represents an empty buffer.
174:
175:      0864                      initBuf:
176:      0864 1803 0905 0900          movw #buffer,bufStart
177:      086A 1803 0905 0902          movw #buffer,bufEnd
178:      0870 79 0904                clr numBuf
179:      0873 3D                    rts
. . .
244:  ;-----
245:  ; Subroutine: chr <- getChBuf
246:  ; Return value
247:  ;      chr - accumulator A - the character read from the buffer
248:  ; Global Variables
249:  ;      numBuf - number of characters in buffer
250:  ;      bufStart - pointer to start of characters in buffer
251:
252:      08AC                      getChBuf
253:      08AC 34                    pshx      ; preserve
254:
255:      08AD F7 0904                tst numBuf
256:      08B0 26 03                  bne unbuf
257:      08B2 87                      clra      ; return 0 null
258:      08B3 20 13                  bra gCBdone
259:
260:      08B5 FE 0900                unbuf    ldx bufStart
261:      08B8 A6 30                  ldaa 1,x+ ; get char, inc bufStart
262:      08BA 73 0904                dec numBuf
263:      08BD 8E 0969                cpx #(buffer+BUFLen)
264:      08C0 2D 03                  blt svBfSt ; bufStart < buffer+BUFLen?
265:      08C2 CE 0905                ldx #buffer
266:      08C5 7E 0900                svBfSt  stx bufStart
267:
268:      08C8 30                      gCBdone pulx      ; restore
269:      08C9 3D                      rts

```

a) (14 points) Given the initial conditions illustrated in Annex A before the following subroutine call (note that the initial conditions correspond to the state of the CPU just before processing an interrupt – the following instruction is the first instruction in the ISR),

```
309:    08EC 16 08AB    sc0_isr jsr getChBuf    ; get char to transmit
```

complete the following table. Complete the values (in hex) in the second column (2 points per value) after the execution of the instruction identified by the line number of the instruction in the previous page.

Line number	Value of registers/memory
253	SP = PC =
255	numBuf =
261	X = A =
262	numBuf =
266	bufStart =

b) (6 points) Complete the following table to show the contents of the stack when the instruction on line 253 has been executed. Show all changes to the stack from the point before the execution of line 309 (i.e. `jsr getChBuf`). Note that the “`jsr getChBuf`” is the first line of an Interrupt service routine (i.e. show how the contents of the stack are affected by an invocation of an interrupt).

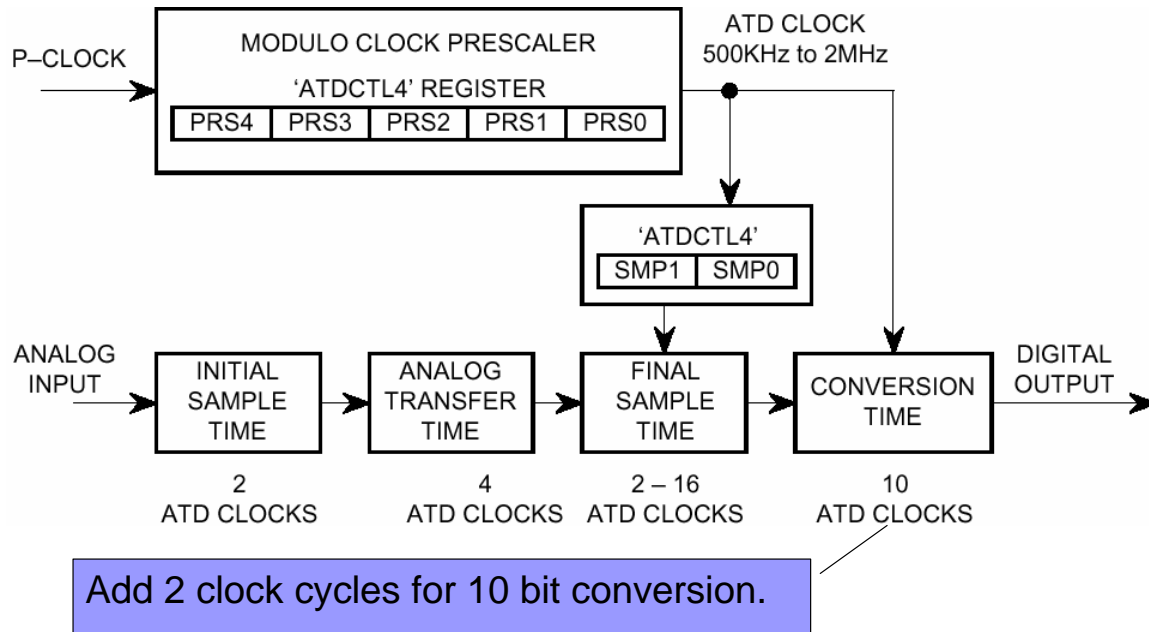
[illegible]

Part 2 Theory Questions (total 20 points)

QUESTION 2.1

(10 points) The diagram below illustrates how the ATD clock is generated and applied to the various steps in converting an analog input to a digital output. Describe the following:

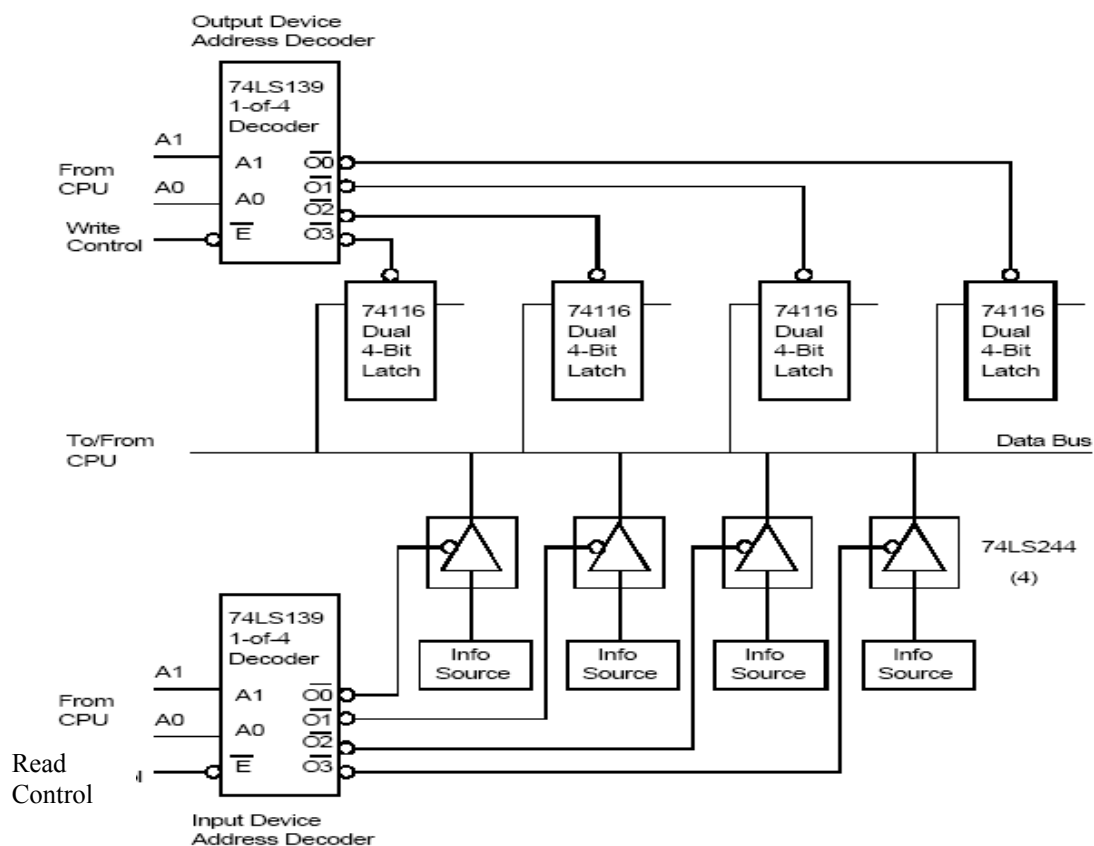
- How the ATD clock is generated from the system P-clock.
- How the frequency of the ATD clock affects the conversion time.
- The relationship between the conversion time and the signal sampling frequency.



QUESTION 2.2

(10 points) The diagram below illustrates how computer system buses are used to perform I/O between the CPU and peripherals.

- Describe the steps taken to read from an input device/register.
- Describe the steps taken to write to an output device/register.
- Explain how the same address can be used for reading from one device/register and writing to another device/register.

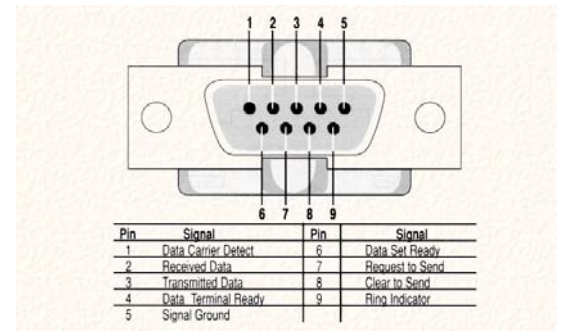
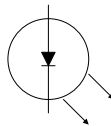
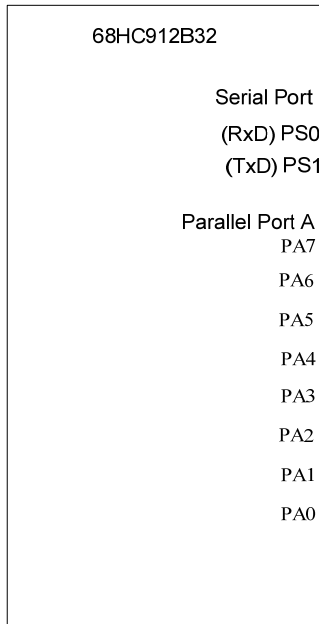


Part 3 – Application Questions (total 60 points)

QUESTION 3.1 (Total 30 points)

This question will be based on the Transmission Driver Module described in Annex B.

- a) **(10 points)** Complete the following circuit that the Transmission Driver Module uses to transmit characters across an EIA-232 interface. Note you only need to make the connections for full duplex communications. It will also light a LED when transmitting data. The LED should be controlled using pin PA0.



b) (8 points) The transmit hardware sub-module is initialized by the following subroutine:

```
; Subroutine: initTxHrd
; Description:
;   Initializes SCI interface
;   Initializes PORTA (to control Transmit LED)

initTxHrd:
    ldd    #sc0_isr          ; Interrupt vector
    pshd
    ldd    #SC0VECT
    jsr    [setUserVector,PCR]
    puld
    ; Init SCI registers
    movw   #208,SC0BDH    ; data rate
    movb   #$12,SC0CR1    ; %0001 0010
    movb   #$08,SC0CR2    ; %0000 1000
    ; Initialise Port A
    movb   $FF,DDRA
    rts
```

Answer the following questions:

b1. What is the purpose of the call to the *setUserVector* subroutine?

b2. After running *initTxHrd*,

(i) What will be the baud rate of the SC0 Interface.?

(ii) How many bits are transmitted between the start and stop bits? How many of these bits are data bits?

c) (12 points total) Provide a design of the ISR *sc0_isr* using a flowchart and then code the ISR. Ensure to properly document both flowchart and code. (Assume that the Tx interrupt is the only interrupt required, i.e., the ISR does not need to handle any additional interrupts)

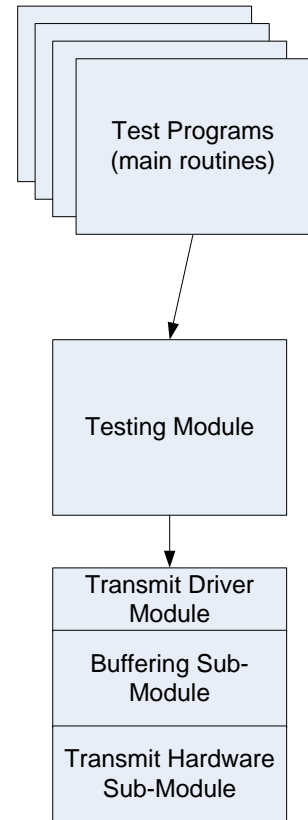
Design (6 points)

Code (6 points)

QUESTION 3.2 (Total 30 points)

A testing module provides support to test the Transmit Driver Module (see adjacent diagram). The testing module contains the following subroutines:

- `initTc2` – initialise the timer and channel 2 to provide delays (no interrupts are used).
- `startDelay` – starts a delay by initialising channel 2.
- `sndTimedMsg(tstMsg)` – subroutine called to send a message (with address `tstMsg` stored in the X register). Note that this subroutine sends the message only if the delay has expired (the idea is to provide the possibility of eventually test a reception module at the same time as the transmission driver module). It will not block, that is, return immediately so that the main routine may continue execution. Thus the routine allows the implementation of polling, that is, send the message only when the delay is complete.



The following is an example of a test program that uses the testing module:

```

;-----Test1 Main Routine-----
; Main Routine: test1
; Description
;   This routine initializes the system with the following characteristics:
;   Every second, transmit the message "Test Message" (with CR,LF)
;   The test is a basic test since it will take less than a second to send the
;   message. It is run in a loop to test the circular buffer function, but will
;   not fill the buffer.

test1:
    lds #STACK                ; Initialize the stack
    jsr initTxDr              ; Initialize Transmit Driver Module
    jsr initTc2               ; Initialize test module
    movb #4,countInit        ; Set to transmit every second
    jsr startDelay
    ldx #simpleMsg             ; Message to transmit
tst1lp jsr sndTimedMsg         ; Poll timer to send message
    bra tst1lp

simpleMsg db "Test Message",CR,LF,$0
  
```

a. (10 points) *initTc2*

The testing module should allow delays to run between $\frac{1}{4}$ second to 1 minute (60 seconds). Select a configuration of the timer and channel 2 that will allow the implementation of such delays (i.e. channel 2 should provide a delay of $\frac{1}{4}$ second). Note that an 8-bit down counter will be implemented by the other subroutines (*startDelay* and *sndTimedMsg*). Explain the selection of your configuration and provide the code for *initTc2*.

a.1. How did you select the configuration of the Timer and Channel 2?

a.2. *initTc2* code:

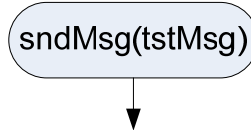
b. (10 points) Implementing delays:

b.1 - Describe how a down counter can be implemented with the timer to achieve delays between $\frac{1}{4}$ second and 1 minute. Describe how a global variable `initCount` can be used to control the length of the delay. (Describe in your own words, do not use flowcharts or code).

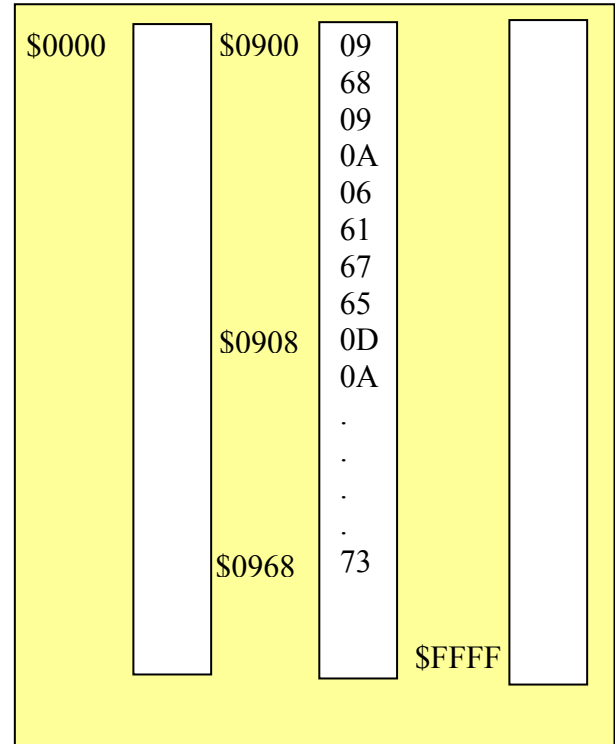
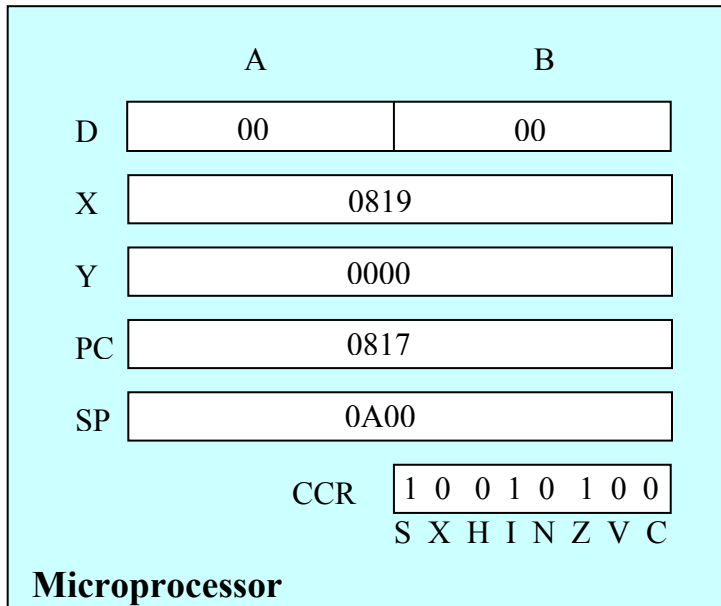
b.2 – Complete the `startDelay` subroutine to initialise the down count.

- c. (10 points) The *sndTimedMsg* subroutine sends a message when the delay has expired by buffering the message using *addStrBuf(strpt)* from the Transmit Driver Module (see Annex B). It has the following features:
- Implements the down count as described in b.1
 - When a down count reaches 0, then add the test string to the character circular buffer using *addStrBuf*. (Note that *addStrBuf* can block when the circular buffer is full).
 - Since the *addStrBuf* can block, start another delay using *startDelay* after buffering a test message.

c.1 Provide a design using a flowchart of the *sndTimedMsg* subroutine.



c.2 Provide the code of the *sndTimedMsg* subroutine. Include documentation in your subroutine.

Annex A - CPU and Memory for Part 1

The initial conditions correspond to the state of the CPU at the moment of processing an interrupt – the following instruction is the first instruction in the ISR.

Addresses of global variables

```

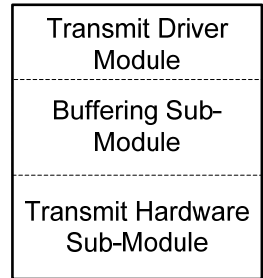
321:      ***** Data for Driver Module *****
322:      =00000900                                ORG DATA
323:
324:      0900 +0002                                bufStart ds 2 ; pointer to start of data
325:      0902 +0002                                bufEnd ds 2 ; pointer to end of data
326:      0904 +0001                                numBuf ds 1 ; number of characters
327:      0905 +0064                                buffer ds BUFLen ; the buffer

```

Annex B

Transmit Driver Module

The Transmit Driver Module contains the code for transmitting via the SCI (SC0) communications interface (asynchronous communication) strings buffered in a circular buffer. As well, a transmission LED (Tx LED) is lit when the module is transmitting data (similar to the Tx LED on a modem).



As shown, this module is subdivided into two sub-modules (each sub-module requires the other to operate):

- The buffering sub-module uses a circular buffer to buffer and unbuffer data. It allows external modules to add strings for transmission to the circular buffer. The Transmit Hardware sub-module will read the data from the buffer (one byte at a time) for transmission via the SC0 interface. A subroutine in this sub-module will enable the TEI interrupt to start (and continue) the transmission of data from the circular buffer. When the interrupt is enabled to start transmission, the Tx LED will also be turned on (see the next page for details on how this sub-module uses the circular buffer).
- The Tx Hardware sub-module provides
 - An initialisation routine to initialise the SC0 and the PORTA (for controlling an LED via pin PA0).
 - An ISR to transmit characters from the circular buffer via the SC0 interface.
 When the buffer is empty, the ISR will disable the TEI interrupt (the interrupt is enabled by the buffering sub-module).

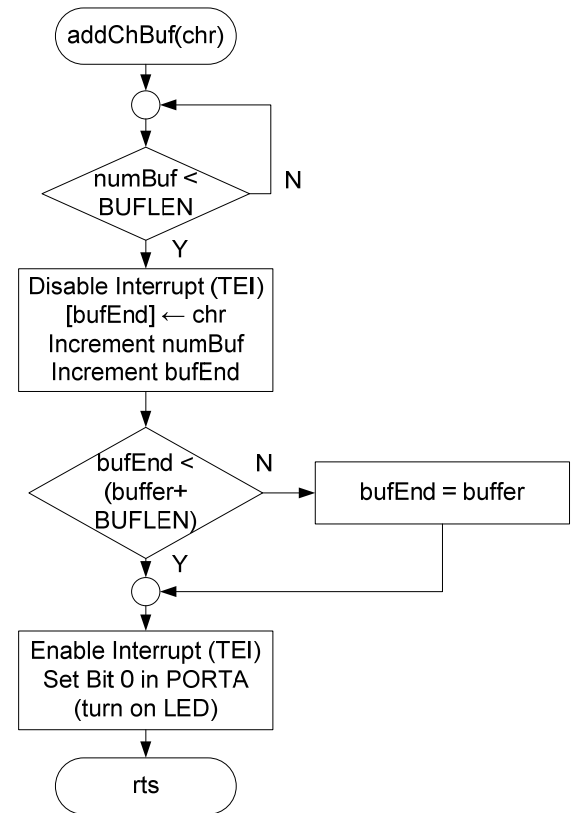
Subroutines defined in the Transmit Driver Module:

- **initTxDr** – initialisation of the Transmit Driver Module. It calls initialisation routines in each of the sub-modules (initBuf and initTxHrd).
- Buffering Sub-module:
 - **initBuf** – Initialises the buffer global variables: bufStart, bufEnd and numBuf.
 - **addStrBuf(strpt)** – Adds a string to the circular buffer. Calls the addChBuf subroutine to buffer each character in the null terminated string addressed by *strpt* (passed in the X register).
 - **addChBuf(chr)** – Adds the character *chr* into the circular buffer. If the buffer is full (numBuf = BUFLen), then the subroutine blocks until space is released by the Transmit Hardware sub-module. This subroutine temporarily disables the TEI interrupt while adding the character to the buffer. Note that re-enabling the interrupt when the buffer was previously empty (i.e. interrupts already disabled) corresponds to starting the transmission of the new content in the buffer.
 - **chr←getChBuf** – This sub-routine returns a character at the head of the circular buffer (via bufStart) or null (0) if the buffer is empty. This subroutine is meant to be called by the ISR to get a character. The *chr* value is returned in Accumulator A.
- Transmit Hardware Sub-Module
 - **initTxHrd** – Initialises the SC0 with speed, frame format, and to process the TEI interrupt (note it does not enable interrupt). Also initialises PORTA to control the Tx LED via bit 0 of PORT A.
 - **sc0_isr** – This is an ISR that will transmit characters found in the circular buffer (using a call to *getChBuf*). If the buffer is empty, it will disable the SC0 interrupt and turn off the Tx LED.

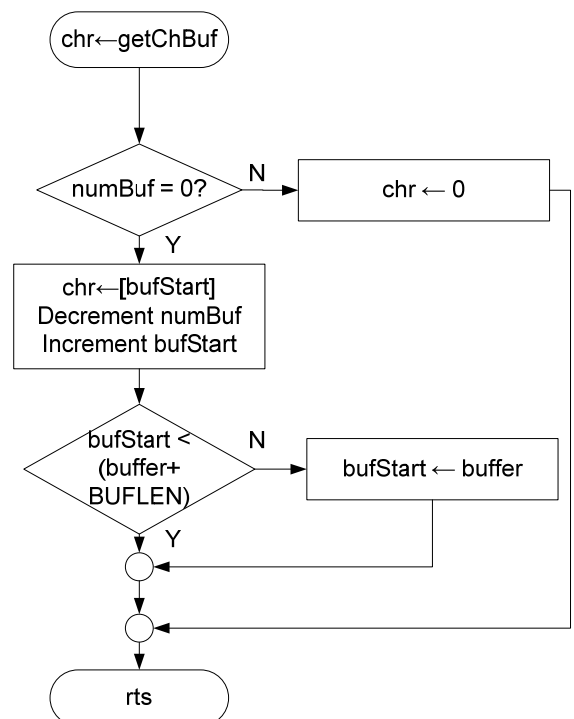
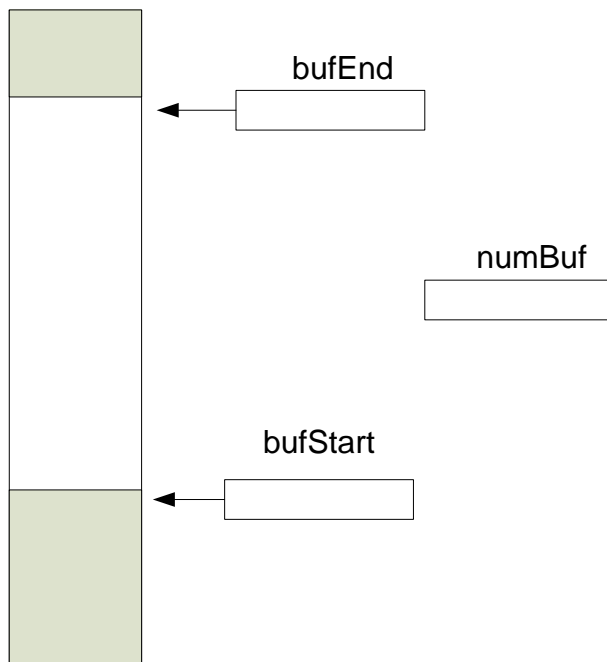
Buffering Sub-Module using a Circular Buffer

The operation of the circular buffer is as follows:

- When a character is added to the buffer, it is added using the *bufEnd* pointer variable. Thus the pointer points to the next available space. After the character is stored, the pointer is incremented as is the variable *numBuf* to indicate the number of characters buffered. The constant *BUFLen* specifies the size of the buffer. The algorithm for *addChBuf* is shown. Note the busy wait when the buffer is full. Also note how the TEI interrupt is disabled while manipulating the buffer global variables.
- A character is removed from the buffer using the *bufStart* pointer variable. This pointer points to the first character buffered. After the character is removed from the buffer, the pointer is incremented and the variable *numBuf* is decremented to reduce the number of characters in the buffer. The algorithm illustrated below for *getChBuf* shows how the character is removed from the buffer.
- The buffer is circular since the buffer pointers will return back to the beginning of the buffer when it reaches its end. The diagram below shows how the data buffer circulates back to the start of the memory allocated to the buffer. The shaded area represents the characters buffered.



buffer (with length BUFLen)



ANNEX C

68HC12 INSTRUCTION LIST (reduced)

Loads, Stores, and Transfers

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Clear Memory Byte	CLR			X	X	X		$m(ea) \leftarrow 0$
Clear Accumulator A (B)	CLRA (B)						X	$A \leftarrow 0$
Load Accumulator A (B)	LDAA (B)	X	X	X	X	X		$A \leftarrow [m(ea)]$
Load Double Accumulator D	LDD	X	X	X	X	X		$D \leftarrow [m(ea, ea+1)]$
Load Effective Address into SP (X or Y)	LEAS (A,B)							$SP \leftarrow ea$
Store Accumulator A (B)	STAA (B)	X	X	X	X	X		$m(ea) \leftarrow (A)$
Store Double Accumulator D	STD	X	X	X	X	X		$m(ea, ea+1) \leftarrow D$
Transfer A to B	TAB						X	$B \leftarrow (A)$
Transfer A to CCR	TAP						X	$CCR \leftarrow (A)$
Transfer B to A	TBA						X	$A \leftarrow B$
Transfer CCR to A	TPA						X	$A \leftarrow (CCR)$
Exchange D with X (Y)	XGDX						X	$D \leftrightarrow (X)$
Pull A (B) from Stack	PULA(B)						X	$A \leftarrow [m(SP)], SP \leftarrow (SP)+1$
Push A (B) onto Stack	PSHA(B)						X	$SP \leftarrow (SP)-1, m(SP) \leftarrow A$

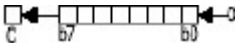

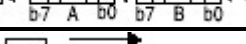
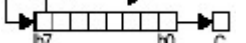
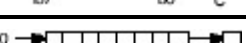
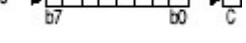
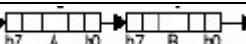
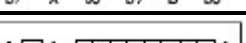
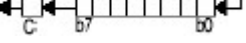
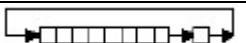
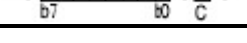
Arithmetic Operations

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Add Accumulators	ABA						X	$A \leftarrow (A) + (B)$
Add with Carry to A (B)	ADCA (B)	X	X	X	X	X		$A \leftarrow (A) + [m(ea)] + (C)$
Add Memory to A (B)	ADDA (B)	X	X	X	X	X		$A \leftarrow (A) + [m(ea)]$
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X		$D \leftarrow (D) + [m(ea, ea+1)]$
Decrement Memory Byte	DEC			X	X	X		$m(ea) \leftarrow [m(ea)] - 1$
Decrement Accumulator A (B)	DECA (B)						X	$A \leftarrow (A) - 1$
Increment Memory Byte	INC			X	X	X		$m(ea) \leftarrow [m(ea)] + 1$
Increment Accumulator A (B)	INCA (B)						X	$A \leftarrow (A) + 1$
Subtract with Carry from A (B)	SBCA (B)	X	X	X	X	X		$A \leftarrow (A) - [m(ea)] - C$
Subtract Memory from A (B)	SUBA (B)	X	X	X	X	X		$A \leftarrow (A) - [m(ea)]$
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X		$D \leftarrow (D) - [m(ea, ea+1)]$
Multiply (byte, unsigned)	MUL						X	$D \leftarrow (A) \times (B)$
Multiply word, unsigned (signed)	EMUL(S)						X	$Y:D \leftarrow (D) \times (Y)$
Unsigned (signed) 32 by 16 divide	EDIV(S)						X	$X \leftarrow (Y:D) / (X), Y \leftarrow \text{quotient}, D \leftarrow \text{remainder}$
Fractional Divide ($D < X$)	FDIV						X	$X \leftarrow (D) / (X), D \leftarrow \text{remainder}$
Integer Divide (unsigned)	IDIV						X	$X \leftarrow (D) / (X), D \leftarrow \text{remainder}$

Logical Operations

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
AND A (B) with Memory	ANDA (B)	X	X	X	X	X		$A \leftarrow A \bullet [m(ea)]$
Bit(s) Test A (B) with Memory	BITA (B)	X	X	X	X	X		$A \bullet [m(ea)]$
One's Complement Memory Byte	COM			X	X	X		$m(ea) \leftarrow \sim [m(ea)]$
One's Complement A (B)	COMA (B)						X	$A \leftarrow \sim A$
OR A (B) with Memory (Exclusive)	EORA (B)	X	X	X	X	X		$A \leftarrow A \oplus [m(ea)]$
OR A (B) with Memory (Inclusive)	ORAA (B)	X	X	X	X	X		$A \leftarrow A + [m(ea)]$

Shift and Rotate

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Arithmetic/Logical Shift Left Memory	ASL			X	X	X		
Arithmetic/Logical Left A (B)	ASLA(B)						X	
Arithmetic/Logical Shift Left Double	ASLD						X	
Arithmetic Shift Right Memory	ASR			X	X	X		
Arithmetic Shift Right A (B)	ASRA(B)						X	
Logical Shift Right A (B)	LSRA(B)						X	
Logical Shift Right Memory	LSR			X	X	X		
Logical Shift Right D	LSRD						X	
Rotate Left (Right) Memory	ROL(R)			X	X	X		
Rotate Left A (B)	ROLA(B)						X	
Rotate Right A (B)	RORA(B)						X	

Compare & Test

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Compare A to B	CBA						X	(A)-(B)
Compare A (B) to Memory	CMPA (B)	X	X	X	X	X		(A) - [m(ea)]
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X		(D) - [m(ea,ea+1)]
Compare SP to Memory (16 Bit)	CPS	X	X	X	X	X		(SP) - [m(ea,ea+1)]
Compare X (Y) to Memory (16 Bit)	CPX	X	X	X	X	X		(X) - [m(ea,ea+1)]
Test memory for 0 or minus	TST			X	X	X		m(ea) - 0
Test A (B) for 0 or minus	TSTA (B)						X	(A)-0

Short Branches

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	PC <= ea if
Branch ALWAYS	BRA		X					
Branch if Carry Clear	BCC		X					C = 0 ?
Branch if Carry Set	BCS		X					C = 1 ?
Branch if Equal Zero	BEQ		X					Z = 1 ?
Branch if Not Equal	BNE		X					Z = 0 ?
Branch if Higher	BHI		X					Unsigned >
Branch if Lower or Same	BLS		X					Unsigned ≤
Branch if Minus	BMI		X					N = 1 ?
Branch if Plus	BPL		X					N = 0 ?
Branch if Bit(s) Clear in Memory Byte	BRCLR			X	X			[m(ea)]•mask=0
Branch if Bit(s) Set in Memory Byte	BRSET			X	X			[m(ea)]•mask=0
Branch if Overflow Clear	BVC		X					V = 0 ?
Branch if Overflow Set	BVS		X					V = 1 ?
Branch if Greater Than or Equal	BGE		X					Signed ≥
Branch if Greater Than	BGT		X					Signed >
Branch if Less Than or Equal	BLE		X					Signed ≤
Branch if Less Than	BLT		X					Signed <
Branch if Higher or Same (same as BCC)	BHS		X					Unsigned ≥
Branch if Lower (same as BCS)	BLO		X					Unsigned <
Branch Never	BRN		X					3-cycle NOP

Long branch mnemonic = L + Short branch mnemonic, e.g.: BRA → LBRA

Loop Primitive Instructions (counter ctr = A, B, or D)

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	Operation
Decrement counter & branch if =0	DBEQ	X						ctr <= (ctr)-1, if (ctr)=0 => PC <= ea
Decrement counter & branch if ≠0	DBNE	X						ctr <= (ctr)-1, if (ctr) ≠0 => PC <= ea
Increment counter & branch if =0	IBEQ	X						ctr <= (ctr)+1, if (ctr)=0 => PC <= ea
Increment counter & branch if ≠0	IBNE	X						ctr <= (ctr)+1, if (ctr) ≠0 => PC <= ea
Test counter & branch if =0	DBEQ	X						if (ctr)=0 => PC <= ea

Subroutine Calls and Returns

Function	Mnemonic	REL	DIR	EXT	IDX	[IDX]	INH	Operation
Branch to Subroutine	BSR	X						SP <= (SP)-2, m(SP) <= (PC), PC <= ea
Jump to Subroutine	JSR		X	X	X	X		SP <= (SP)-2, m(SP) <= (PC), PC <= ea
CALL a Subroutine (expanded memory)	CALL		X	X	X	X		SP <= (SP)-2, m(SP) <= (PC), PC <= ea SP <= (SP)-1, m(SP) <= (PPG), PC <= pg
Return from Subroutine	RTS						X	PC <= [m(SP)], SP <= (SP)+2
Return from call	RTC						X	PPG <= [m(SP)], SP <= (SP)+1, PC <= [m(SP)], SP <= (SP)+2

Function	Mnemonic	DIR	EXT	IDX	[IDX]	INH	Operation
Jump	JMP	X	X	X	X		PC <= ea

The **jump** instruction allows control to be passed to any address in the 64-Kbyte memory map.

Stack and Index Register Instructions

Function	Mnemonic	IMM	DIR	EXT	IDX	[IDX]	INH	Operation
Decrement Index Register X (Y)	DEX (Y)						X	$X \leftarrow (X) - 1$
Increment Index Register X (Y)	INX (Y)						X	$X \leftarrow (X) + 1$
Load Index Register X (Y)	LDX(Y)	X	X	X	X	X		$X \leftarrow [m(ea,ea+1)]$
Pull X (Y) from Stack	PULX						X	$X \leftarrow [m(SP,SP+1)]$ $SP \leftarrow (SP) + 2$
Push X (Y) onto Stack	PSHX (Y)						X	$m(SP,SP+1) \leftarrow (X)$ $SP \leftarrow (SP) - 2$
Store Index Register X (Y)	STX (X)	X	X	X	X	X		$m(ea,ea+1) \leftarrow X$
Add Accumulator B to X (Y)	ABX (Y)						X	$X \leftarrow (X) + (B)$
Decrement Stack Pointer	DES						X	$SP \leftarrow (SP) - 1$
Increment Stack Pointer	INS						X	$SP \leftarrow (SP) + 1$
Load Stack Pointer	LDS	X	X	X	X	X		$SP \leftarrow [m(ea,ea+1)]$
Store Stack Pointer	STS	X	X	X	X	X		$m(ea,ea+1) \leftarrow (SP)$
Transfer SP to X (Y)	TSX (Y)						X	$X \leftarrow (SP)$
Transfer X (Y) to SP	TXS (Y)						X	$SP \leftarrow (X)$
Exchange D with X (Y)	XGDX (Y)						X	$(D) \leftrightarrow (X)$

Function	Mnemonic	INH	Operation
Return from Interrupt	RTI	X	$(M_{(SP)} \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$
Software Interrupt	SWI	X	
Wait for Interrupt	WAI	X	

Interrupt Handling

The software interrupt (SWI) instruction is similar to a JSR instruction, except the contents of all working CPU registers are saved on the stack rather than just the return address. SWI is unusual in that it is requested by the software program as opposed to other interrupts that are requested asynchronously to the executing program

Annex D - Register and Bit Definitions for 68HC12 Peripheral Modules

DATA REGISTERS								CONTROL REGISTERS								STATUS REGISTERS							
PARALLEL PORTS								DDRA \$0002 DDRB \$0003 DDRE \$0009 DDRT \$00AF															
PORTA \$0000 PORTB \$0001 PORTE \$0008 PORTT \$00AE								DDRA7 DDRA6 DDRA5 DDRA4 DDRA3 DDRA2 DDRA1 DDRA0															
PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0								DDRB7 DDRB6 DDRB5 DDRB4 DDRB3 DDRB2 DDRB1 DDRB0															
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0								DDRE7 DDRE6 DDRE5 DDRE4 DDRE3 DDRE2 DDRE1 DDRE0															
PE7 PE6 PE5 PE4 PE3 PE2 PE1 PE0								DDRT7 DDRT6 DDRT5 DDRT4 DDRT3 DDRT2 DDRT1 DDRT0															
PT7 PT6 PT5 PT4 PT3 PT2 PT1 PT0								PUCR \$000C															
								0 0 0 PUPE 0 0 PUPB PUPA															
TIMER								TIOS \$0080 Timer								TFLG1 \$008E Timer Interrupt Flag Register 1							
TCNT \$0084 \$0085 Timer Count Register								Input-Capture (IOS=0)/Output-Compare (IOS=1) Select Register								C7F C6F C5F C4F C3F C2F C1F C0F							
Bit 15 Bit 14 Bit 13 Bit 12 Bit 11 Bit 10 Bit 9 Bit 8								IOS7 IOS6 IOS5 IOS4 IOS3 IOS2 IOS1 IOS0								TFLG2 \$008E Timer Interrupt Flag 2							
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0								CFORC \$0081 Timer Compare Force Register								TOF 0 0 0 0 0 0 0 0							
								FOC7 FOC6 FOC5 FOC4 FOC3 FOC2 FOC1 FOC0															
								OC7M \$0082 Timer Output Compare 7 Mask															
								OC7M7 OC7M6 OC7M5 OC7M4 OC7M3 OC7M2 OC7M1 OC7M0															
								OC7D \$0083 Timer Ouput Compare 7 Data Register															
								OC7D7 OC7D6 OC7D5 OC7D4 OC7D3 OC7D2 OC7D1 OC7D0															
								TCTL1/TCTL2 \$0088/89 Timer Control Register 1/2															
								OM7 OL7 OM6 OL6 OM5 OL5 OM4 OL4															
								OM3 OL3 OM2 OL2 OM1 OL1 OM0 OL0															
								TCTL3/TCTL4 \$008A/8B Timer Control Register 4															
								EDG7B EDG7A EDG6B EDG6A EDG5B EDG5A EDG4B EDG4A															
								EDG3B EDG3A EDG2B EDG2A EDG1B EDG1A EDG0B EDG0A															
								TSCR \$86 Timer System Control Register															
								TEN TFFCA															
								TMSK1 \$008D Timer Interrupt Mask 2 Register															
								C7I C6I C5I C4I C3I C2I C1I C0I															
								TMSK2 \$008C Timer Interrupt Mask 1 Register															
								TOI 0 PUPT RDPT TCRE PR2 PR1 PR0															
SERIAL INTERFACE-SCI								SC0BDH.\$00C0 \$00C1 SCI Baud Rate Control Register								Baud Rate SER PT=0 => even							
SC0DRH \$00C6 SCI Data Register High								0 0 0 SER12 SER11 SER10 SER9 SER8								2400 208 M=0 => 1 start bit, 8 data bits, 1 stop bit							
R8 T8 0 0 0 0 0 0								SER7 SER6 SER5 SER4 SER3 SER2 SER1 SER0								9600 52 RDRF = 0 => SC0DR is empty							
SC0DRL \$00C7 SCI Data Register Low								SC0CR1 \$00C2 SCI Control Register 1								14,400 35 \$FFD6 = SCI0 interrupt vector address							
R7T7 R6T6 R5T5 R4T4 R3T3 R2T2 R1T1 R0T0								LOOPS WOMS RSRC M WAKE ILT PE PT								19,200 26							
								SC0CR2 \$00C3 SCI Control Register 2								SC0SR1 \$00C4 SCI Status Register 1							
								TIE TCIE RIE ILIE TE RE RWU SBK								TDRE TC RDRF IDLE OR NF FE PF							
ANALOG-TO-DIGITAL SUBSYSTEM								ATDCTL2 \$62 ATD Control Register 2								ATDSTAT \$66 \$67 ATD Status Register 0,1							
ADRxH \$70+2x ATD Result Registers High								ADPU AFFC AWAI 0 0 0 ASCIE ASCIF								SCF 0 0 0 0 CC2 CC1 CC0							
Bit 15 Bit 14 Bit 13 Bit 12 Bit 11 Bit 10 Bit 9 Bit 8								ATDCTL4 \$64 ATD Control Register 4								CCF7 CCF6 CCF5 CCF4 CCF3 CCF2 CCF1 CCF0							
ADRxL: \$71+2x ATD Result Registers Low								S10BM SMP1 SMP0 PRS4 PRS3 PRS2 PRS1 PRS0															
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0								ATDCTL5 \$65 ATD Control Register 5															
x=0...7								N/A S8CM SCAN MULT CD CC CE CA															
								INTCR \$001E Interrupt Control Register								HPRIO \$001F Highest Priority I Interrupt Register							
								IRQE IRQEN DLY 0 0 0 0 0 0								1 1 PSEL5 PSEL4 PSEL3 PSEL2 PSEL1 0							

Timer Module

- TEN - Activate (=0) and deactivate (=1) timer
- AFFC - allows automatic clear of flags (=1)
- TOI enable (=1) overflow interrupt (from \$FFFF to \$0000) of the TCNT
- PUPT enable (=1) input pin pullup resistors
- RDPT enable output pin power reduction (=1)
- PR2, PR1, PR0 prescale bits ($= 2^{PR}$) applied to system clock for incrementing timer
- IOS0 to IOS7 – Configure channel as output compare (=1) or input-capture (=0)
- FOC0-FOC1 – Force output-compare event on channel
- OC7M0-OC7M7 – Allow channel 7 to affect channel pin (=1)
- OC7D0-OC7D7 – Value of level for channel 7 to output to channel pin
- C0I-C7I – Enable a channel interrupt
- C0F-C7F – Channel flag set when event occurs
- TOF – Counter Overflow flag

ATD Module

- ADPU enable (=1) converter
- AFFC enable (=1) fast clear
- ASCIE enable (=1) interruption at the completion of a conversion sequence
- ASCFI flags (=1) a sequence conversion
- S10BM selects resolution (0 = 8 bits, 1 = 10 bits)
- SMP1, SMP2 – sample time ($= 2^{SMP+1}$)
- PRS4, PRS3, PRS2, PRS1, PRS0 – prescaler of system clock (PC – typically 8 MHz) to generate converter clock (CC) : $CC = (PC/(PRS+1))/2$, where PRS is value represented by PRS4 to PRS0.
- SC8CM – selects 4 conversions (=0) or 8 conversions (=1) in a sequence.
- SCAN – single conversion (=0) or scan mode (=1)
- MULT – single pin (=0) or multiple pins (=1)
- CD, CD, CB, CA – Defines the pin(s) for conversion.

Serial Interface Module – SCI subsystem

- M: Two modes 8 data bits or 9 data bits (with 1 start bit and 1 stop bit)
- PE, PT: Enable parity (PE) and type of parity (PT – 0 for even)
- TIE, TCIE, RIE: Bits for enabling interruptions for reception and transmission
- TE, RE: Enable transmission (TE) and reception (RE)
- TDRE: 1 indicates that more data can be written into the data register
- TC: 1 indicates that transmission of data is complete
- RDRF: 1 indicates that reception data register is full
- OR, NF, FE, PF: error bits, equal 1 when an error is detected during reception (overrun, noise, framing, parity)
- RAF: 1 indicates that reception of a character is underway.

ATD Clock Scaling

Prescale Bits PRS4-PRS0	Divisor	ATD Clock
00000	Not used	-
00001	4 (default)	2 MHz
00010	6	1.33 MHz
00011	8	1 MHz
00100	10	800 kHz
00101	12	667 kHz
00110	14	571 kHz
00111	16	500 kHz
01xxx 11xxx	Not used	

Annex E - MAX562 from MAXIM

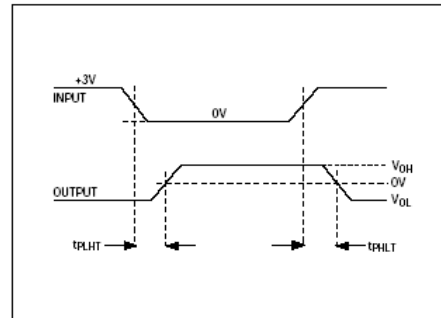
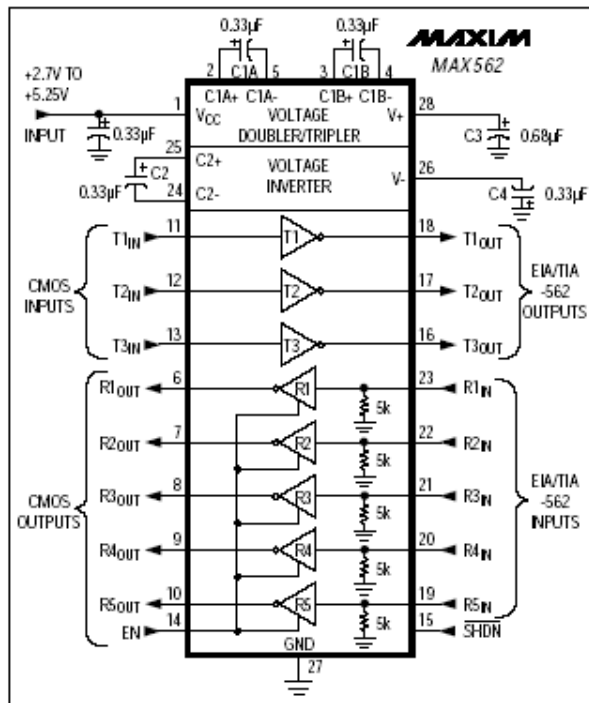


Figure 1. Transmitter Propagation Delay Timing

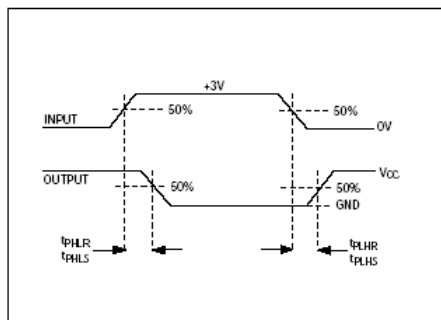


Figure 2. Receiver Propagation Delay Timing