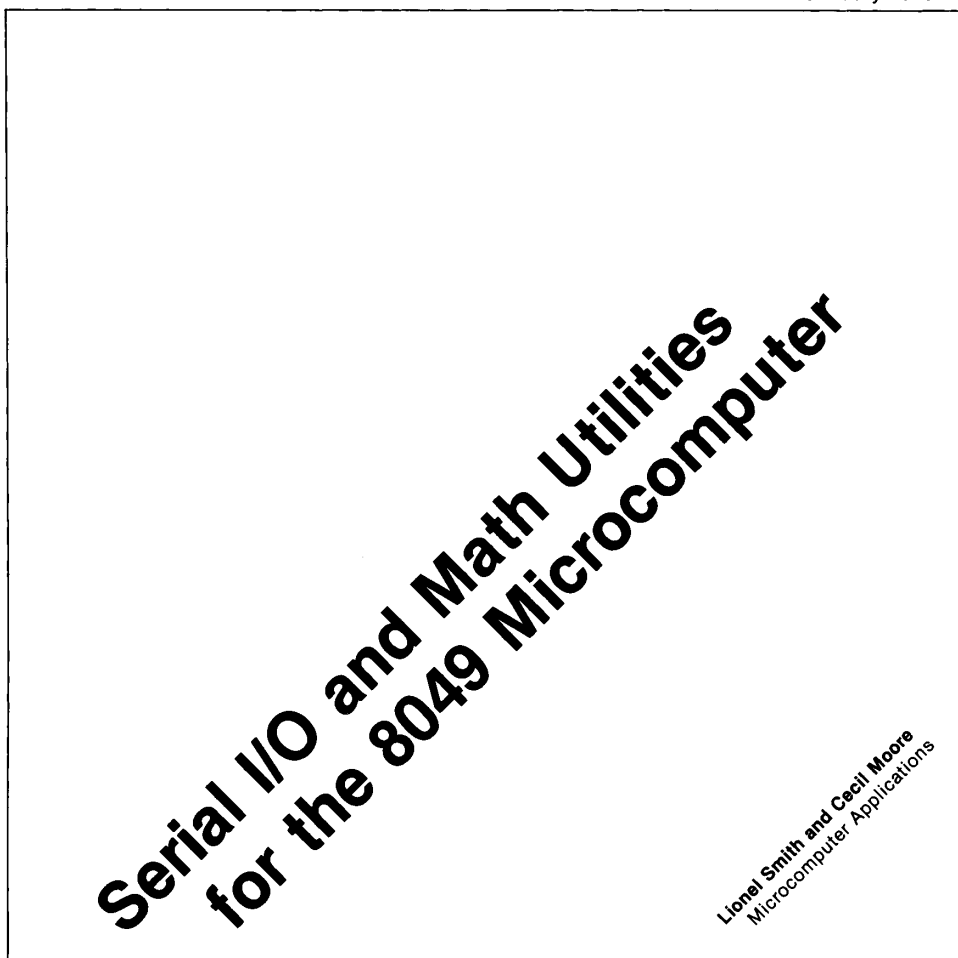




January 1979



© Intel Corporation 1979

9800904



INTRODUCTION

The Intel® MCS-48 family of microcomputers marked the first time an eight bit computer with program storage, data storage, and I/O facilities was available on a single LSI chip. The performance of the initial processors in the family (the 8748 and the 8048) has been shown to meet or exceed the requirements of most current applications of microcomputers. A new member of the family, however, has been recently introduced which promises to allow the use of the single chip microcomputer in many application areas which have previously required a multichip solution. The Intel® 8049 virtually doubles processing power available to the systems designer. Program storage has been increased from 1K bytes to 2K bytes, data storage has been increased from 64 bytes to 128 bytes, and processing speed has been increased by over 80%. (The 2.5 microsecond instruction cycle of the first members of the family has been reduced to 1.36 microseconds.)

It is obvious that this increase in performance is going to result in far more ambitious programs being written for execution in a single chip microcomputer. This article will show how several program modules can be designed using the 8049. These modules were chosen to illustrate the capability of the 8049 in frequently encountered design situations. The modules included are full duplex serial I/O, binary multiply and divide routines, binary to BCD conversions, and BCD to binary conversion. It should be noted that since the 8049 is totally software compatible with the 8748 and 8048 these routines will also be useful directly on these processors. In addition the algorithms for these programs are expressed in a program design language format which should allow them to be easily understood and extended to suit individual applications with minimal problems.

FULL DUPLEX SERIAL COMMUNICATIONS

Serial communications have always been an important facet in the application of microprocessors. Although this has been partially due to the necessity of connecting a terminal to the microprocessor based system for program generation and debug, the main impetus has been the simple fact that a large share of microprocessors find their way into end products (such as intelligent terminals) which themselves depend on serial communication. When it is necessary to add a serial link to a microprocessor such as the Intel® MCS-85 or 86 the solution is easy; the Intel® 8251A USART or 8273 SDLC chip can easily be added to provide the necessary protocol. When it is necessary to do the same thing to a single chip microcomputer, however, the situation becomes more difficult.

Some microcomputers, such as the Intel 8048 and 8049 have a complete bus interface built into them which allows the simple connection of a USART to the processor chip. Most other single chip microcomputers, although lacking such a bus, can be connected to a USART with various artificial hardware and software constructs. The difficulty with using these chips,

however, is more economic than technical; these same peripheral chips which are such a bargain when coupled to a microprocessor such as the MCS-85 or 86, have a significant cost impact on a single chip microcomputer based system. The high speed of the 8049, however, makes it feasible to implement a serial link under software control with no hardware requirements beyond two of the I/O pins already resident on the microcomputer.

There are many techniques for implementing serial I/O under software control. The application note "Application Techniques for the MCS-48 Family" describes several alternatives suitable for half duplex operation. Full duplex operation is more difficult, however, since it requires the receive and transmit processes to operate concurrently. This difficulty is made more severe if it is necessary for some other process to also operate while serial communication is occurring. Scanning a keyboard and display, for example, is a common operation of single chip microcomputer based system which might have to occur concurrently with the serial receive/transmit process. The next section will describe an algorithm which implements full duplex serial communication to occur concurrently with other tasks. The design goal was to allow 2400 baud, full duplex, serial communication while utilizing no more than 50% of the available processing power of the high speed 8049 microcomputer.

The format used for most asynchronous communication is shown in Figure 1. It consists of eight data bits with a leading 'START' bit and one or more trailing 'STOP' bits. The START bit is used to establish synchronization between the receiver and transmitter. The STOP bits ensure that the receiver will be ready to synchronize itself when the next start bit occurs. Two stop bits are normally used for 110 baud communication and one stop bit for higher rates.

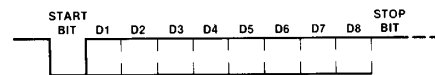


Figure 1.

The algorithm used for reception of the serial data is shown in Figure 2. It uses the on board timer of the 8049 to establish a sampling period of four times the desired baud rates. For 2400 baud operation a crystal frequency of 9.216 MHz was chosen after the following calculation:

$$f = 480N(2400)(4)$$

- where 480 is the factor by which the crystal frequency is divided within the processor to get the basic interrupt rate
- 2400 is the desired baud rate
- 4 is the required number of samples per bit time
- N is the value loaded into the MCS-48 timer when it overflows

The value N was chosen to be two (resulting in $f = 9.216$ MHz) so that the operating frequency of the 8049 could be as high as possible without exceeding the maximum frequency specification of the 8049 (11 MHz).

```

;
;
; START OF RECEIVE ROUTINE
; =====
;
;1 IF RECEIVE FLAG=0 THEN
;2   IF SERIAL INPUT=SPACE THEN
;3     RECEIVE FLAG:=1
;3     BYTE FINISHED FLAG:=0
;2   ENDIF
;1 ELSE   SINCE RECEIVE FLAG=1 THEN
;2   IF SYNC FLAG=0 THEN
;3     IF SERIAL INPUT=SPACE THEN
;4       SYNC FLAG:=1
;4       DATA:=80H
;4       SAMPLE CNTR:=4
;3     ELSE   SINCE SERIAL INPUT=MARK THEN
;4       RECEIVE FLAG:=0
;3     ENDIF
;2   ELSE   SINCE SYNC FLAG=1 THEN
;3     SAMPLE COUNTER:=SAMPLE COUNTER-1
;3     IF SAMPLE COUNTER=0 THEN
;4       SAMPLE COUNTER:=4
;4       IF BYTE FINISHED FLAG=0 THEN
;5         CARRY:=SERIAL INPUT
;5         SHIFT DATA RIGHT WITH CARRY
;5         IF CARRY=1 THEN
;6           OKDATA:=DATA
;6           IF DATA READY FLAG=0 THEN
;7             BYTE FINISHED FLAG:=1
;6           ELSE
;7             BYTE FINISHED FLAG:=1
;7             OVERRUN FLAG:=1
;6           ENDIF
;5         ENDIF
;4       ELSE   SINCE BYTE FINISHED FLAG=1 THEN
;5         IF SERIAL INPUT=MARK THEN
;6           DATA READY FLAG:=1
;5         ELSE   SINCE SERIAL INPUT=SPACE THEN
;6           ERROR FLAG:=1
;5         ENDIF
;5         RECEIVE FLAG:=0
;5         SYNC FLAG:=0
;4       ENDIF
;3     ENDIF
;2   ENDIF
;1 ENDIF
    
```

Figure 2

The timer interrupt service routine always loads the timer with a constant value. In effect the timer is used to generate an independent time base of four times the required baud rate. This time base is free running and is never modified by either the receive or transmit programs, thus allowing both of them to use the same timer. Routines which do other time dependent tasks (such as scanning keyboards) can also be called periodically at some fixed multiple of this basic time unit.

The algorithm shown in Figure 2 uses this basic clock plus a handful of flags to process the serial input data.

Once the meaning of these flags are understood the operation of the algorithm should be clear. The **Receive Flag** is set whenever the program is in the process of receiving a character. The **Sync Flag** is set when the center of the start bit has been checked and found to be a SPACE (if a MARK is detected at this point the receiver process has been triggered by a noise pulse so the program clears the **Receive Flag** and returns to the idle state). When the program detects synchronization it loads the variable **DATA** with 80H and starts sampling the serial line every four counts. As the data is received it is right shifted into variable **DATA**; after eight bits have been received the initial one set into **DATA** will result in a carry out and the program knows that it has received all eight bits. At this point it will transfer all eight bits to the variable **OKDATA** and set the **Byte Finished Flag** so that on the next sample it will test for a valid stop bit instead of shifting in data. If this test is successful the **Data Ready Flag** will be set to indicate that the data is available to the main process. If the test is unsuccessful the **Error Flag** will be set.

The transmit algorithm is shown in Figure 3. It is executed immediately following the receive process. It is a simple program which divides the free running clock down and transmits a bit every fourth clock. The variable **TICK COUNTER** is used to do the division. The **Transmitting Flag** indicates when a character transmission is in progress and is also used to determine when the START bit should be sent. The **TICK COUNTER** is used to determine when to send the next bit ($TICK\ COUNTER\ MODULO\ 4 = 0$) and also when the STOP bits should be sent ($TICK\ COUNTER = 9\ 4$). After the transmit routine completes any other timer based routines, such as a keyboard/display scanner or a real time clock, can be executed.

```

;
; START OF TRANSMIT ROUTINE
; =====
;
;1
;1 TICK COUNTER:=TICK COUNTER+1
;1 IF TICK COUNTER MOD 4=0 THEN
;2   IF TRANSMITTING FLAG=1 THEN
;3     IF TICK COUNTER=00 1010 00 BINARY THEN
;4       TRANSMITTING FLAG:=0
;3     ELSE   IF TICK COUNTER=00 1001 00 BINARY THEN
;4       SEND END MARK
;4       TRANSMITTING FLAG:=0
;3     ELSE   SINCE TICK COUNTER>THE ABOVE COUNT THEN
;4       SEND NEXT BIT
;3     ENDIF
;2   ELSE   SINCE TRANSMITTING FLAG=0 THEN
;3     IF TRANSMIT REQUEST FLAG=1 THEN
;4       XMTBYT:=RXTBYT
;4       TRANSMIT REQUEST FLAG:=0
;4       TRANSMITTING FLAG:=1
;4       TICK COUNTER:=0
;4       SEND SYNC BIT (SPACE)
;3     ENDIF
;2   ENDIF
;1 ENDIF
    
```

Figure 3

Figure 4 shows the complete receive and transmit programs as they are implemented in the instruction set of

the 8049. Also included in Fig. 4 is a short routine which was used to test the algorithm.

ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V2.0

```

LOC  OBJ      SEQ      SOURCE STATEMENT
          1 :*****
          2 ;*
          3 ;*   THIS PROGRAM TESTS THE FULL DUPLEX COMMUNICATION SOFTWARE
          4 ;*
          5 :*****
          6 ;
          7 $INCLUDE(<F1.URTEST.POL)
          8 ;
          9 ;   START OF TEST ROUTINE
         10 ;   =====
         11 ;
         12 ;
         13 ;
         14 ;
         15 ;
         16 ;1 ERROR COUNT =0
         17 ;1 REPEAT
         18 ;2   PATTERN =0
         19 ;2   INITIALIZE TIMER
         20 ;2   CLEAR FLAGBYTE
         21 ;2   FLAG1=MARK
         22 ;2   REPEAT
         23 ;3   IF TRANSMIT REQUEST FLAG=0 THEN
         24 ;4       N$TBYTE =PATTERN
         25 ;4       TRANSMIT REQUEST FLAG=1
         26 ;3   ENDIF
         27 ;3   IF DATA READY FLAG=1 THEN
         28 ;4       PATTERN =0$DATA
         29 ;4       DATA READY FLAG =0
         30 ;3   ENDIF
         31 ;2   UNTIL ERROR FLAG OR OVERRUN FLAG
         32 ;2   INCREMENT ERROR COUNT
         33 ;1 UNTIL FOREVER
         34 ;EOF
         35 ;EJECT
0000          36   ORG   0
         37 ;1 SELECT REGISTER BANK 0
0000 C5          38   SEL   RB0
         39 ;1 GOTO TEST
0001 2400          40   JMP   TEST
         41 $   INCLUDE(<F1.UART)
         42 ;
         43 ;
         44 ;   ASYNCHRONOUS RECEIVE/TRANSMIT ROUTINE
         45 ;   =====
         46 ;   THIS ROUTINE RECEIVES SERIAL CODE USING PIN P1N 0 AS RXD
         47 ;   AND CONCURRENTLY TRANSMITS USING PIN P27
         48 ;   NOTE
         49 ;   THIS ROUTINE USES FLAG 1 TO BUFFER THE TRANSMITTED

```

Figure 4

All mnemonics copyrighted © Intel Corporation 1979.

```

LOC  OBJ      SEQ      SOURCE STATEMENT
= 50 ; 1 DATA LINE. THIS ELIMINATES THE JITTER THAT
= 51 ; 1 WOULD BE CAUSED BY VARIATIONS IN THE RECEIVE
= 52 ; 1 TIMING. NO OTHER PROGRAM MAY USE FLAG 1 WHILE
= 53 ; 1 THE TIMER INTERRUPT IS ENABLED
= 54 ;
= 55 ;
= 56 ;
= 57 ;
= 58 ;
= 59 ;      REGISTER ASSIGNMENTS-BANK1
= 60 ;      =====
= 61 ;
= 62 ;
0007    = 63 ATEMP EQU    R7      ; USED TO SAVE ACCUMULATOR CONTENTS DURING INTERRUPT
0006    = 64 FLGBYT EQU    R6      ; CONTAINS VARIOUS FLAGS USED TO CONTROL THE RECEIVE
= 65 ;      AND TRANSMIT PROCESS. SEE CONSTANT DEFINITIONS FOR
= 66 ;      THE MEANING OF EACH BIT
0005    = 67 SAMCTR EQU    R5      ; SAMPLE COUNTER FOR THE RECIEVE PROCESS
0004    = 68 TCKCTR EQU    R4      ; SAMPLE COUNTER FOR THE TRANSMIT PROCESS
0000    = 69 REG00 EQU    R0      ; USED AS POINTER REGISTER
= 70 ;
= 71 ;      PAM ASSIGNMENTS
= 72 ;      =====
= 73 ;
0020    = 74 M0KDAT EQU    20H     ; RECEIVE RETURNS VALID DATA IN THIS BYTE
0021    = 75 M0DATA EQU    21H     ; RECEIVE ACCUMULATES DATA IN THIS BYTE
0022    = 76 M0NXTBY EQU    22H     ; CONTAINS BYTE BEING TRANSMITTED
0023    = 77 M0NXTBY EQU    23H     ; CONTAINS THE NEXT BYTE TO BE TRANSMITTED
= 78 $EJECT
= 79 ;
= 80 ;
= 81 ;      CONSTANTS
= 82 ;      =====
= 83 ;
= 84 ;      THE FOLLOWING CONSTANTS ARE USED TO ACCESS THE FLAG BITS CONTAINED
= 85 ;      IN REGISTER FLGBYT
= 86 ;
0001    = 87 RCVFLG EQU    01H     ; SET WHEN START BIT IS FIRST DETECTED
= 88 ;      ; RESET WHEN RECEIVE PROCESS IS COMPLETE
0002    = 89 SYNFLG EQU    02H     ; SET WHEN START BIT IS VERIFIED
= 90 ;      ; RESET WHEN RECEIVE PROCESS IS COMPLETE
0004    = 91 BYFNFL EQU    04H     ; RESET WHEN START BIT IS FIRST DETECTED
= 92 ;      ; SET WHEN THE EIGHT DATA BITS HAVE ALL BEEN RECEIVED
0008    = 93 DRDYFL EQU    08H     ; SHOULD BE RESET BY MAIN PROGRAM WHEN DATA IS ACCEPTED
= 94 ;      ; SET BY RECEIVE PROCESS WHEN STOP BIT(S) ARE VERIFIED
0010    = 95 ERRFLG EQU    10H     ; SHOULD BE RESET BY MAIN PROGRAM WHEN SAMPLED
= 96 ;      ; SET BY RECEIVE PROCESS IF A FRAMING ERROR IS DETECTED
0020    = 97 TRROFL EQU    20H     ; TESTED BY MAIN PROGRAM TO DETERMINE IF READY TO
= 98 ;      ; TRANSMIT A NEW BYTE-SET TO INDICATE THAT NXTBYT
= 99 ;      ; HAS BEEN LOADED
= 100 ;      ; RESET BY TRANSMIT PROCESS WHEN BYTE IS ACCEPTED
0040    = 101 TRNGFL EQU    40H     ; SET WHEN TRANSMISSION OF A BYTE STARTS
= 102 ;      ; RESET WHEN STOP BIT IS TRANSMITTED
0080    = 103 OVRUN EQU    80H     ; SET BY RECEIVE PROCESS WHEN OVERUN OCCURS
= 104 ;      ; SHOULD BE RESET BY MAIN PROGRAM WHEN SAMPLED
    
```

Figure 4 (continued)

```

LOC  OBJ      SEQ      SOURCE STATEMENT
      = 105 ;
      = 106 ;      GENERAL CONSTANTS
      = 107 ;      =====
      = 108 ;
0000      = 109 MARK EQU  80H      ; USED TO GENERATE A MARK
FF7F      = 110 SPACE EQU  NOT 80H ; USED TO GENERATE A SPACE
0000      = 111 STPBTS EQU  0      ; CONTROLS THE NUMBER OF STOP BITS
      = 112 ;      ; 0 GENERATES ONE STOP BIT
      = 113 ;      ; 1 GENERATES TWO STOP BITS
      = 114 ;
      = 115 #EJECT
      = 116 ;
      = 117 ;      START OF RECEIVE/TRANSMIT INTERRUPT SERVICE ROUTINE
      = 118 ;      =====
      = 119 ;
0007      = 120 ORG  0007H
      = 121 ;
      = 122 ;1 ENTER INTERRUPT MODE
0007 160A      = 123 TISR: JTF  UART
0009 93      = 124 RETR
000A 05      = 125 UART: SEL  RB1
      = 126 ;1 SAVE ACCUMULATOR CONTENTS
000B AF      = 127 MOV  ATEMP,A
      = 128 ;1 RELOAD TIMER
000C 23FE      = 129 MOV  A,#TIMCNT
000E 62      = 130 MOV  T,A
      = 131 ;
      = 132 ;      OUTPUT TXD BUFFER (F1) TO TXD I/O LINE (P2?)
      = 133 ;      =====
      = 134 ;
000F 7615      = 135 JF1  OMARK
0011 9A7F      = 136 OSPACE: ANL  P2,#SPACE
0013 0417      = 137 JMP  RCV000
0015 8A80      = 138 OMARK ORL  P2,#MARK
      = 139 ;
      = 140 ;      START OF RECEIVE ROUTINE
      = 141 ;      =====
      = 142 ;
      = 143 ;1 IF RECEIVE FLAG=0 THEN
0017 FE      = 144 RCV000: MOV  A,FLGBYT
0018 1224      = 145 JB0  RCV010
      = 146 ;2 IF SERIAL INPUT=SPACE THEN
001A 3664      = 147 JTB  XMIT
      = 148 ;3 RECEIVE FLAG:=1
001C FE      = 149 MOV  A,FLGBYT
001D 4301      = 150 ORL  A,#RCVFLG
      = 151 ;3 BYTE FINISHED FLAG:=0
001F 53FB      = 152 ANL  A,#NOT BVFNFL
      = 153 ;2 ENDF
0021 AE      = 154 MOV  FLGBYT,A
0022 0464      = 155 JMP  XMIT
      = 156 ;1 ELSE SINCE RECEIVE FLAG=1 THEN
      = 157 ;2 IF SYNC FLAG=0 THEN
0024 3238      = 158 RCV010: JBI  RCV030
      = 159 ;3 IF SERIAL INPUT=SPACE THEN

```

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

LOC	OBJ	SEQ	SOURCE STATEMENT
0026	3633	= 160	JT0 RCV020
		= 161 :4	SYNC FLAG:=1
0028	4302	= 162	ORL A,#SYNFLG
002A	AE	= 163	MOV FLGBYT,A
		= 164 :4	DATA:=80H
002B	B821	= 165	MOV R0,#MDATA
002D	B080	= 166	MOV @R0,#80H
		= 167 :4	SAMPLE CNTR:=4
002F	BD04	= 168	MOV SAMCTR,#4
0031	0464	= 169	JMP XMIT
		= 170 :3	ELSE SINCE SERIAL INPUT=MARK THEN
		= 171 :4	RECEIVE FLAG:=0
0033	53FE	= 172 RCV020	ANL A,#NOT RCVFLG
		= 173 :3	ENDIF
0035	AE	= 174	MOV FLGBYT,A
0036	0464	= 175	JMP XMIT
		= 176 :2	ELSE SINCE SYNC FLAG=1 THEN
		= 177 :2	SAMPLE COUNTER:=SAMPLE COUNTER-1
0038	ED64	= 178 RCV030	DJNZ SAMCTR,XMIT
		= 179 :3	IF SAMPLE COUNTER=0 THEN
		= 180 :4	SAMPLE COUNTER:=4
003A	B004	= 181	MOV SAMCTR,#4
		= 182 :4	IF BYTE FINISHED FLAG=0 THEN
003C	5259	= 183	JB2 RCV050
003E	97	= 184	CLR C
		= 185 :5	CARRY:=SERIAL INPUT
003F	2642	= 186	JNT0 RCV040
0041	A7	= 187	CPL C
0042	B821	= 188 RCV040	MOV R0,#MDATA
0044	F0	= 189	MOV A,@R0
		= 190 :5	SHIFT DATA RIGHT WITH CARRY
0045	67	= 191	RRC A
0046	A0	= 192	MOV @R0,A
		= 193 :5	IF CARRY=1 THEN
0047	E664	= 194	JNC XMIT
		= 195 :6	OKDATA:=DATA
0049	B820	= 196	MOV R0,#OKDAT
004B	A0	= 197	MOV @R0,A
		= 198 :6	IF DATA READY FLAG=0 THEN
004C	FE	= 199	MOV A,FLGBYT
004D	7254	= 200	JB3 RCV045
		= 201 :7	BYTE FINISHED FLAG=1
004F	4304	= 202	ORL A,#BYFNFL
0051	AE	= 203	MOV FLGBYT,A
0052	0464	= 204	JMP XMIT
		= 205 :6	ELSE
		= 206 :7	BYTE FINISHED FLAG:=1
		= 207 :7	OVERRUN FLAG:=1
		= 208 RCV045	
		= 209	MOV A,FLGBYT
0054	4304	= 210	ORL A,#(BYFNFL OR OVRUN)
0056	AE	= 211	MOV FLGBYT,A
		= 212 :6	ENDIF
		= 213 :5	ENDIF
0057	0464	= 214	JMP XMIT

Figure 4 (continued)

```

LOC  OBJ      SEQ      SOURCE STATEMENT
      = 215 ;4      ELSE   SINCE BYTE FINISHED FLAG=1 THEN
      = 216 ;5      IF SERIAL INPUT=MARK THEN
0059 265F      = 217 RCV050: JNT0   RCV060
      = 218 ;6      DATA READY FLAG:=1
005B 4308      = 219   ORL   A,#DRDYFL
005D 0461      = 220   JMP   RCV070
      = 221 ;5      ELSE   SINCE SERIAL INPUT=SPACE THEN
      = 222 ;6      ERROR FLAG:=1
005F 4310      = 223 RCV060: ORL   A,#ERRFLG
      = 224 ;5      ENDIF
      = 225 ;5      RECEIVE FLAG:=0
      = 226 ;5      SYNC FLAG:=0
0061 53FC      = 227 RCV070: ANL   A,#NOT(SYNFLG OR RCVFLG)
0063 0E      = 228   MOV   FLGBYT,A
      = 229 ;4      ENDIF
      = 230 ;3      ENDIF
      = 231 ;2      ENDIF
      = 232 ;1      ENDIF
      = 233 #EJECT
      = 234 ;
      = 235 ;      START OF TRANSMIT ROUTINE
      = 236 ;      =====
      = 237 ;
      = 238 ;1
      = 239 ;      ; TRANSMITTER OUTPUT BIT IS P2-7
0064 1C      = 240 ;1 TICK COUNTER =TICK COUNTER+1
      = 241 XMIT: INC   TCKCTR
      = 242 ;1 IF TICK COUNTER MOD 4=0 THEN
0065 2303      = 243   MOV   A,#03H
0067 5C      = 244   ANL   A,TCKCTR
0068 9697      = 245   JNZ   RETURN
      = 246 ;2      IF TRANSMITTING FLAG=1 THEN
006A FE      = 247   MOV   A,FLGBYT
006B 37      = 248   CPL   A
006C 0286      = 249   JB6   XMT040
      = 250   IF STPBTS EQ 1
      = 251 ;3      IF TICK COUNTER=00 1010 00 BINARY THEN
      = 252   MOV   A,#28H      ; CONDITIONAL ASSEMBLY
      = 253   XRL   A,TCKCTR      ;
      = 254   JNZ   XMT010      ;
      = 255 ;4      TRANSMITTING FLAG:=0
      = 256   MOV   A,FLGBYT      ;
      = 257   ANL   A,#NOT TRNGFL ;
      = 258   MOV   FLGBYT,A      ;
      = 259   JMP   RETURN      ;
      = 260   ENDIF
      = 261 ;3      ELSE   IF TICK COUNTER=00 1001 00 BINARY THEN
006E 2324      = 262 XMT010: MOV   A,#24H
0070 0C      = 263   XRL   A,TCKCTR
0071 967B      = 264   JNZ   XMT020
      = 265 ;4      SEND END MARK
0073 05      = 266   CLR   F1      ; SET FLAG1 TO MARK
0074 65      = 267   CPL   F1
      = 268   IF STPBTS EQ 0
      = 269 ;4      TRANSMITTING FLAG:=0

```

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

LOC	OBJ	SEQ	SOURCE STATEMENT
0075	FE	= 270	MOV A, FLGBYT ; CONDITIONAL ASSEMBLY
0076	53BF	= 271	ANL A, #NOT TRNGFL ;
0078	AE	= 272	MOV FLGBYT, A ;
0079	0497	= 273	JMP RETURN ;
		= 274	ENDIF
		= 275 ;3	ELSE SINCE TICK COUNTER<THE ABOVE COUNT THEN
		= 276 ;4	SEND NEXT BIT
007B	B822	= 277 XMT020:	MOV R0, #XMTBY
007D	F0	= 278	MOV A, @R0
007E	67	= 279	RRC A
007F	A0	= 280	MOV @R0, A
0080	A5	= 281	CLR F1 ; FLAG 1 WILL BE USED TO BUFFER TXD
0081	E697	= 282	JNC RETURN ; GO TO RETURN POINT IF TXD=SPACE (0)
0083	B5	= 283	CPL F1 ; ELSE COMPLEMENT FLAG 1 TO A MARK
0084	0497	= 284	JMP RETURN
		= 285 ;3	ENDIF
		= 286 ;2	ELSE SINCE TRANSMITTING FLAG=0 THEN
		= 287 ;3	IF TRANSMIT REQUEST FLAG=1 THEN
0086	B297	= 288 XMT040:	JBS RETURN ; FLAG BYTE THERE
		= 289 ;4	XMTBYT:=NXTBYT
0088	B823	= 290	MOV R0, #NXTBY
008A	F0	= 291	MOV A, @R0
008B	B822	= 292	MOV R0, #XMTBY
008D	A0	= 293	MOV @R0, A
		= 294 ;4	TRANSMIT REQUEST FLAG:=0
008E	FE	= 295	MOV A, FLGBYT
008F	53DF	= 296	ANL A, #NOT TRNGFL
		= 297 ;4	TRANSMITTING FLAG:=1
0091	4340	= 298	ORL A, #TRNGFL
0093	AE	= 299	MOV FLGBYT, A
		= 300 ;4	TICK COUNTER:=0
0094	8C00	= 301	MOV TCKCTR, #0
		= 302 ;4	SEND SYNC BIT (SPACE)
0096	A5	= 303	CLR F1 ; SET FLAG 1 TO CAUSE A SPACE
		= 304 ;3	ENDIF
		= 305 ;2	ENDIF
		= 306 ;1	ENDIF
		= 307	RETURN:
		= 308 ;1	RESTORE ACCUMULATOR
0097	FF	= 309	MOV A, ATEMP
0098	93	= 310	RETR
		311	#EJECT
		312 ;	
		313 ;	START OF TEST ROUTINE
		314 ;	=====
		315 ;	
0100		316	ORG 0100H
FFFE		317 TIMCNT	EQU -2
001E		318 MFLGBY	EQU 1EH
001D		319 M\$AMCT	EQU 1DH
001C		320 MTCKCT	EQU 1CH
		321 ;	
0007		322 ERRCNT	EQU R7
0006		323 PATT	EQU R6
		324 ;	

Figure 4 (continued)

LOC	OBJ	SEG	SOURCE STATEMENT
			325 ;
			326 ;
			327 ;1 ERROR COUNT:=0
0100	BF00		328 TEST: MOV ERRCNT, #0
			329 ;1 REPEAT
			330 TLOP:
			331 ;2 PATTERN:=0
0102	BE00		332 MOV PATT, #00
			333 ;2 INITIALIZE TIMER
0104	23FE		334 MOV A, #TIMCNT
0106	62		335 MOV T, A
0107	55		336 STRT T
0108	25		337 EN TCNTI
			338 ;2 CLEAR FLAGBYTE
0109	B81E		339 MOV R0, #MFLGBY
010B	B000		340 MOV @R0, #0
			341 ;2 FLAG1=MARK
010D	A5		342 CLR F1
010E	B5		343 CPL F1
			344 ;2 REPEAT
			345 TILOP: .
			346 ;3 IF TRANSMIT REQUEST FLAG=0 THEN
010F	B81E		347 MOV R0, #MFLGBY
0111	F0		348 MOV A, @R0
0112	B224		349 JBS TREC
			350 ;4 NXTBYTE:=PATTERN
0114	B923		351 MOV R1, #NXTBY
0116	FE		352 MOV A, PATT
0117	A1		353 MOV @R1, A
			354 ;4 TRANSMIT REQUEST FLAG=1
0118	35		355 DIS TCNTI ; LOCK OUT TIMER INTERRUPT
			356 ; SO THAT MUTUAL EXCLUSION IS MAINTAINED WHILE
			357 ; THE FLAG BYTE IS BEING MODIFIED
0119	F0		358 MOV A, @R0
011A	4320		359 ORL A, #TRRQFL
011C	A0		360 MOV @R0, A
011D	25		361 EN TCNTI
011E	1622		362 JTF TESTA
0120	2424		363 JMP TREC
0122	140A		364 TESTA: CALL UART ; CALL UART BECAUSE TIMER OVERFLOWED DURING LOCKOUT
			365 ;3 ENDIF
			366 ;3 IF DATA READY FLAG=1 THEN
			367 TREC:
0124	F0		368 MOV A, @R0
0125	37		369 CPL A
0126	7238		370 JBS TREC
			371 ;4 PATTERN:=OKDATA
0128	B920		372 MOV R1, #MOKDAT
012A	F1		373 MOV A, @R1
012B	AE		374 MOV PATT, A
			375 ;4 DATA READY FLAG:=0
012C	35		376 DIS TCNTI ; LOCK OUT TIMER INTERRUPT
			377 ; SO THAT MUTUAL EXCLUSION IS MAINTAINED WHILE
			378 ; THE FLAG BYTE IS BEING MODIFIED
012D	F0		379 MOV A, @R0

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

```

LOC  OBJ      SEQ      SOURCE STATEMENT

012E 53F7      380      ANL     A, #NOT DRDYFL
0130 00        381      MOV     @R0, A
0131 25        382      EN     TCNTI
0132 1636      383      JTF     TESTB
0134 2438      384      JMP     TREC
0136 140A      385      TESTB: CALL  UART      ; CALL UART IF TIMER OVERFLOWED DURING LOCKOUT
386      TREC:
387      ;3      ENDIF
388      ;2      UNTIL ERROR FLAG OR OVERRUN FLAG
0138 F0        389      MOV     A, @R0
0139 5390      390      ANL     A, #(OVRUN OR ERRFLG)
013B C60F      391      JZ     TLOP
392      ;2      INCREMENT ERROR COUNT
013D 1F        393      INC     ERRCNT
394      ;1      UNTIL FOREVER
013E 2402      395      JMP     TLOP
396      ;EOF
397      ;END

```

USER SYMBOLS

AEMP 0007	BYFNFL 0004	DRDYFL 0008	ERRCNT 0007	ERRFLG 0010	FLGBYT 0006	MARK 0000	NDATA 0021
MFLGBY 001E	MXMTBY 0023	MXKDAT 0020	MSANCT 001D	NTCKCT 001C	MXMTBY 0022	OMARK 0015	OSPACE 0011
OVRUN 0080	PATT 0006	RCV000 0017	RCV010 0024	RCV020 0033	RCV030 0038	RCV040 0042	RCV045 0054
RCV050 0059	RCV060 005F	RCV070 0061	RCVFLG 0001	REG0 0000	RETURN 0097	SANCTR 0005	SPACE FF7F
STPBTS 0000	SYNFLG 0002	TCKCTR 0004	TEST 0100	TESTA 0122	TESTB 0136	TLOP 010F	TIMCNT FFFE
TISR 0007	TLOP 0102	TREC 0124	TREC 0138	TRNGFL 0040	TRR0FL 0020	UART 000A	XMI1 0064
XMT010 006E	XMT020 007B	XMT040 0086					

ASSEMBLY COMPLETE, NO ERRORS

Figure 4 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

MULTIPLY ALGORITHMS

Most microcomputer programmers have at one time or another implemented a multiply routine as part of a larger program. The usual procedure is to find an algorithm that works and modify it to work on the machine being used. There is nothing wrong with this approach. If engineers felt that they had to reinvent the wheel every time a new design is undertaken, that's probably what most of us would be doing—designing wheels. If the efficiency of the multiply algorithm, either in terms

of code size or execution time is important, however, it is necessary to be reasonably familiar with the multiplication process so that appropriate optimizations for the machine being used can be made.

To understand how multiplication operates in the binary number system, consider the multiplication of two four bit operands A and B. The "ones and zeros" in A and B represent the coefficients of two polynomials. The operation A × B can be represented as the following multiplication of polynomials:

$$\begin{array}{r}
 \begin{array}{r}
 A3 \cdot 2^3 + A2 \cdot 2^2 + A1 \cdot 2^1 + A0 \cdot 2^0 \\
 \times B3 \cdot 2^3 + B2 \cdot 2^2 + B1 \cdot 2^1 + B0 \cdot 2^0
 \end{array} \\
 \hline
 B0A3 \cdot 2^3 + B0A2 \cdot 2^2 + B0A1 \cdot 2^1 + B0A0 \cdot 2^0 \\
 + B1A3 \cdot 2^4 + B1A2 \cdot 2^3 + B1A1 \cdot 2^2 + B1A0 \cdot 2^1 \\
 + B2A3 \cdot 2^5 + B2A2 \cdot 2^4 + B2A1 \cdot 2^3 + B2A0 \cdot 2^2 \\
 + B3A3 \cdot 2^6 + B3A2 \cdot 2^5 + B3A1 \cdot 2^4 + B3A0 \cdot 2^3
 \end{array}$$



The sum of all these terms represents the product of A and B. The simplest multiply algorithm factors the above terms as follows:

$$A * B = B0 * (A) * 2^0 + B1 * (A) * 2^1 + B2 * (A) * 2^2 + B3 * (A) * 2^3$$

Since the coefficients of B (i.e., B0, B1, B2, and B3) can only take on the binary values of 1 or 0, the sum of the products can be formed by a series of simple adds and multiplications by two. The simplest implementation of this would be:

```
MULTIPLY:
  PRODUCT = 0
  IF B0 = 1 THEN PRODUCT := PRODUCT + A
  IF B1 = 1 THEN PRODUCT := PRODUCT + 2 * A
  IF B2 = 1 THEN PRODUCT := PRODUCT + 4 * A
  IF B3 = 1 THEN PRODUCT := PRODUCT + 8 * A
  END MULTIPLY
```

In order to conserve memory, the above straight line code is normally converted to the following loop:

```
MULTIPLY:
  PRODUCT := 0
  COUNT := 4
  REPEAT
  IF B[0] = 1 THEN PRODUCT := PRODUCT + A
  A := 2 * A
  B := B / 2
  COUNT := COUNT - 1
  UNTIL COUNT = 0
  END MULTIPLY
```

The repeated multiplication of A by two (which can be performed by a simple left shift) forms the terms 2 * A, 4 * A, and 8 * A. The variable B is divided by two (performed by a simple right shift) so that the least significant bit can always be used to determine whether the addition should be executed during each pass through the loop. It is from these shifting and addition opera-

tions that the "shift and add" algorithm takes its common name.

The "shift and add" algorithm shown above has two areas where efficiency will be lost if implemented in the manner shown. The first problem is that the addition to the partial product is double precision relative to the two operands. The other problem, which is also related to double precision operations, is that the A operand is double precision and that it must be left shifted and then the B operand must be right shifted. An examination of the "longhand" polynomial multiplication will reveal that, although the partial product is indeed double precision, each addition performed is only single precision. It would be desirable to be able to shift the partial product as it is formed so that only single precision additions are performed. This would be especially true if the partial product could be shifted into the "B" operand since one bit of the partial product is formed during each pass through the loop and (happily) one bit of the "B" operand is vacated. To do this, however, it is necessary to modify the algorithm so that both of the shifts that occur are of the same type.

To see how this can be done one can take the basic multiplication equation already presented:

$$A * B = B0 * (A * 2^0) + B1 * (A * 2^1) + B2 * (A * 2^2) + B3 * (A * 2^3)$$

and factoring 2⁴ from the right side:

$$A * B = 2^4 [B0 * (A * 2^{-4}) + B1 * (A * 2^{-3}) + B2 * (A * 2^{-2}) + B3 * (A * 2^{-1})]$$

This operation has resulted in a term (within the brackets) which can be formed by right shifts and adds and then multiplied by 2⁴ to get the final result. The resulting algorithm, expanded to form an eight by eight multiplication, is shown in figure 5. Note that although the result is a full sixteen bits, the algorithm only performs eight bit additions and that only a single sixteen bit shift operation is involved. This has the effect of reducing both the code space and the execution time for the routine.

ISIS-II MCS-48/UP1-41 MACRO ASSEMBLER, V2.0

LOC	OBJ	SEQ	SOURCE STATEMENT
		1	\$MACROFILE
		2	\$INCLUDE(:F1.MPYS.HED)
		3	*****
		4	;
		5	;
		6	MPY8X8
		7	*****
		8	;
		9	THIS UTILITY PROVIDES AN 8 BY 8 UNSIGNED MULTIPLY
		10	AT ENTRY:
		11	A = LOWER EIGHT BITS OF DESTINATION OPERAND
		12	XA= DON'T CARE
		13	R1= POINTER TO SOURCE OPERAND (MULTIPLIER) IN INTERNAL MEMORY

Figure 5

All mnemonics copyrighted © Intel Corporation 1979.

LOC	OBJ	SEQ	SOURCE STATEMENT
		= 14 ;*	*
		= 15 ;*	AT EXIT. *
		= 16 ;*	A = LOWER EIGHT BITS OF RESULT *
		= 17 ;*	XA= UPPER EIGHT BITS OF RESULT *
		= 18 ;*	C = SET IF OVERFLOW ELSE CLEARED *
		= 19 ;*	*
		= 20 ;*****	*****
		21 ;	
		22 ;	
		23 \$INCLUDE(<:F1:MPY8.PDL)	
		= 24 ;1 MPY8X8	
		= 25 ;1 MULTIPLICAND(15-8)=0	
		= 26 ;1 COUNT=#8	
		= 27 ;1 REPEAT	
		= 28 ;2 IF MULTIPLICAND(8)=0 THEN BEGIN	
		= 29 ;3 MULTIPLICAND:=MULTIPLICAND/2	
		= 30 ;2 ELSE	
		= 31 ;3 MULTIPLICAND(15-8):=MULTIPLICAND(15-8)+MULTIPLIER	
		= 32 ;3 MULTIPLICAND:=MULTIPLICAND/2	
		= 33 ;2 ENDF	
		= 34 ;2 COUNT=COUNT-1	
		= 35 ;1 UNTIL COUNT=0	
		= 36 ;1 END MPY8X8	
		37 ;	
		38 EQUATES	
		39 ; =====	
		40 ;	
0002		41 XA EQU R2	
0002		42 COUNT EQU R3	
0004		43 ICNT EQU R4	
		44 ;	
0003		45 DIGPR EQU 3	
		46 ;	
		47 \$EJECT	
		48 \$INCLUDE(<:F1:MPY8)	
		= 49 ;1 MPY8X8:	
		= 50 MPY8X8:	
		= 51 ;1 MULTIPLICAND(15-8):=0	
0000 8A00		= 52 MOV XA,#00	
		= 53 ;1 COUNT=#8	
0002 8E00		= 54 MOV COUNT,#8	
		= 55 ;1 REPEAT	
		= 56 MPY8SLP:	
		= 57 ;2 IF MULTIPLICAND(8)=0 THEN BEGIN	
0004 120E		= 58 JBO MPY8A	
		= 59 ;3 MULTIPLICAND:=MULTIPLICAND/2	
0006 20		= 60 XCH A,XA	
0007 97		= 61 CLR C	
0008 67		= 62 RRC A	
0009 2A		= 63 XCH A,XA	
000A 67		= 64 RRC A	
000B EB04		= 65 D.INZ COUNT,MPY8SLP	
000D 83		= 66 RET	
		= 67 ;2 ELSE	

Figure 5 (continued)

All mnemonics copyrighted © Intel Corporation 1979.



```

LOC  OBJ      SEQ      SOURCE STATEMENT
      = 68 MPY8A
      = 69 :3      MULTIPLICAND(15-8):=MULTIPLICAND(15-8)+MULTIPLIER
000E 2A      = 70      XCH  A, XA
000F 61      = 71      ADD  A, @R1
0010 67      = 72      RRC  A
0011 2A      = 73      XCH  A, XA
0012 67      = 74      RRC  A
0013 EB04    = 75      DJNZ  COUNT, MPY8LP
0015 83      = 76      RET
      = 77 :3      MULTIPLICAND:=MULTIPLICAND/2
      = 78 :2      ENDIF
      = 79 :2      COUNT:=COUNT-1
      = 80 :1 UNTIL COUNT=0
      = 81 :1 END MPY8X8
      = 82  END

USER SYMBOLS
COUNT 0003  DIGPF 0003  ICNT  0004  MPY8A 000E  MPY8LP 0004  MPY8X8 0000  XA  0002

ASSEMBLY COMPLETE. NO ERRORS
    
```

All mnemonics copyrighted © Intel Corporation 1979.

DIVIDE ALGORITHMS

In order to understand binary division a four bit operation will again be used as an example. The following algorithm will perform a four by four division:

```

DIVIDE:
IF 16*DIVISOR>= DIVIDEND THEN
SET OVERFLOW ERROR FLAG
ELSE
IF 8*DIVISOR>= DIVIDEND THEN
QUOTIENT(3):= 1
DIVIDEND:= DIVIDEND - 8*DIVISOR
ELSE
QUOTIENT(3):= 0
ENDIF
IF 4*DIVISOR>= DIVIDEND THEN
QUOTIENT(2):= 1
DIVIDEND:= DIVIDEND - 4*DIVISOR
ELSE
QUOTIENT(2):= 0
ENDIF
IF 2*DIVISOR>= DIVIDEND THEN
QUOTIENT(1):= 1
DIVIDEND:= DIVIDEND - 2*DIVISOR
ELSE
QUOTIENT(1):= 0
ENDIF
IF 1*DIVISOR>= DIVIDEND THEN
QUOTIENT(0):= 1
DIVIDEND:= DIVIDEND - 1*DIVISOR
ELSE
QUOTIENT(0):= 0
ENDIF
ENDIF
END DIVIDE
    
```

The algorithm is easy to understand. The first test asks if the division will fit into the dividend sixteen times. If it will, the quotient cannot be expressed in only four bits so an overflow error flag is set and the divide algorithm ends. The algorithm then proceeds to determine if eight times the divisor fits, four times, etc. After each test it either sets or clears the appropriate quotient bit and modifies the dividend. To see this algorithm in action, consider the division of 15 by 5:

00001111	(15)	
- 01010000	(16*5)	
Doesn't fit—no overflow		
00001111	(15)	
- 00101000	(8*5)	
Doesn't fit—Q[3]= 0		
00001111	(15)	
- 00010100	(4*5)	
Doesn't fit—Q[2]= 0		
00001111	(15)	
- 00001010	(2*5)	
Fits—Q[1]= 1		
00000101	(15-2*5)	
- 00000101	(1*5)	
Fits—Q[0]= 1		
00000000		

The result is Q = 0011 which is the binary equivalent of 3—the correct answer. Clearly this algorithm can (and has been) converted to a loop and used to perform divisions. An examination of the procedure, however, will show that it has the same problems as the original multiply algorithm.

The first problem is that double precision operations are involved with both the comparison of the division with the dividend and the conditional subtraction. The second problem is that as the quotient bits are derived they must be shifted into a register. In order to reduce the register requirements, it would be desirable to shift them into the divisor register as they are generated since the divisor register gets shifted anyway. Unfortunately the quotient bits are derived most significant bits first so doing this will form a mirror image of the quotient—not very useful.

Both of these problems can be solved by observing that the algorithm presented for divide will still work if both sides of all the “equations” involving the dividend are divided by sixteen. The looping algorithm then would proceed as follows:

```
DIVIDE:
QUOTIENT:= 0
COUNT:= 4
DIVIDEND:= DIVIDEND/16
IF DIVISOR>= DIVIDEND THEN
  OVERFLOW FLAG:= 1
ELSE
  REPEAT
    DIVIDEND:= DIVIDEND*2
    QUOTIENT:= QUOTIENT*2
    IF DIVISOR>= DIVIDEND THEN
      QUOTIENT:= QUOTIENT + 1/*SET QUOTIENT[0]*/
      DIVIDEND:= DIVIDEND - DIVISOR
    ENDIF
    COUNT:= COUNT - 1
  UNTIL COUNT= 0
ENDIF
END DIVIDE
```

When this algorithm is implemented on a computer which does not have a direct compare instruction the comparison is done by subtraction and the inner loop of the algorithm is modified as follows:

```
*
*
REPEAT
  DIVIDEND:= DIVIDEND*2
  QUOTIENT:= QUOTIENT*2
  DIVIDEND:= DIVIDEND - DIVISOR
  IF BORROW= 0 THEN
    QUOTIENT:= QUOTIENT + 1
  ELSE
    DIVIDEND:= DIVIDEND + DIVISOR
  ENDIF
  COUNT:= COUNT - 1
UNTIL COUNT= 0
*
*
```

An implementation of this algorithm using the 8049 instruction set is shown in figure 6. This routine does an unsigned divide of a 16 bit quantity by an eight bit quantity. Since the multiply algorithm of figure 5 generates a 16 bit result from the multiplication of two eight bit operands, these two routines complement each other and can be used as part of more complex computations.

IS15-II M.S-48/UPI-41 MACRO ASSEMBLER, V2.0

LOC	OBJ	SEQ	SOURCE STATEMENT
		1	\$MACROFILE
		2	\$INCLUDE('F1-DIV16.HED')
=		3	*****
=		4	;* *
=		5	;* DIV16 *
=		6	;* *
=		7	*****
=		8	;* *
=		9	;* THIS UTILITY PROVIDES AN 16 BY 8 UNSIGNED DIVIDE *
=		10	;* AT ENTRY: *
=		11	;* A = LOWER EIGHT BITS OF DESTINATION OPERAND *
=		12	;* %A= UPPER EIGHT BITS OF DIVIDEND *
=		13	;* R1= POINTER TO DIVISOR IN INTERNAL MEMORY *
=		14	;* *
=		15	;* AT EXIT: *
=		16	;* A = LOWER EIGHT BITS OF RESULT *
=		17	;* %A= REMAINDER *

Figure 6

LOC	OBJ	SEQ	SOURCE STATEMENT
		= 18 ;*	C = SET IF OVERFLOW ELSE CLEARED *
		= 19 ;*	*
		= 20 ;	*****
		= 21 ;	
		= 22 ;	
		= 23 ;	#INCLUDE(F1:DIV16.PDL)
		= 24 ;1	DIV16
		= 25 ;1	COUNT=#8
		= 26 ;1	DIVIDEND(15-8)=DIVIDEND(15-8)-DIVISOR
		= 27 ;1	IF BORROW=0 THEN /* IT FITS*/
		= 28 ;2	SET OVERFLOW FLAG
		= 29 ;1	ELSE
		= 30 ;2	RESTORE DIVIDEND
		= 31 ;2	REPEAT
		= 32 ;3	DIVIDEND:=DIVIDEND*2
		= 33 ;3	QUOTIENT:=QUOTIENT*2
		= 34 ;3	DIVIDEND(15-8)=DIVIDEND(15-8)-DIVISOR
		= 35 ;3	IF BORROW=1 THEN
		= 36 ;4	RESTORE DIVIDEND
		= 37 ;3	ELSE
		= 38 ;4	QUOTIENT(0)=1
		= 39 ;3	ENDIF
		= 40 ;3	COUNT=COUNT-1
		= 41 ;2	UNTIL COUNT=0
		= 42 ;2	CLEAR OVERFLOW FLAG
		= 43 ;1	ENDIF
		= 44 ;1	ENDDIVIDE
		= 45 ;	
		= 46 ;	EQUATES
		= 47 ;	=====
		= 48 ;	
0002		= 49 XA	EQU R2
0003		= 50 COUNT	EQU R3
		= 51 ;	
		= 52 ;	#EJECT
		= 53 ;	#INCLUDE(F1:DIV16)
		= 54 ;1	DIV16:
0000 2A		= 55 DIV16:	XCH A,XA ; ROUTINE WORKS MOSTLY WITH BITS 15-8
		= 56 ;1	COUNT=#8
0001 B808		= 57	MOV COUNT,#8
		= 58 ;1	DIVIDEND(15-8)=DIVIDEND(15-8)-DIVISOR
0003 37		= 59	CPL A
0004 61		= 60	ADD A,@R1
0005 37		= 61	CPL A
		= 62 ;1	IF BORROW=0 THEN /* IT FITS*/
0006 F60B		= 63	JC DIV1A
		= 64 ;2	SET OVERFLOW FLAG
0008 A7		= 65	CPL C
0009 0424		= 66	JMP DIV1B
		= 67 ;1	ELSE
		= 68 DIV1A:	
		= 69 ;2	RESTORE DIVIDEND
000B 61		= 70	ADD A,@R1
		= 71 ;2	REPEAT
		= 72 DIV1B:	
		= 73 ;3	DIVIDEND:=DIVIDEND*2

Figure 6 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

LOC	OBJ	SEQ	SOURCE STATEMENT
		= 74 :3	QUOTIENT =QUOTIENT*2
0000	97	= 75	CLR C
0000	2A	= 76	NCH A,XA
000E	F7	= 77	RLO A
000F	2A	= 78	NCH A,XA
0010	F7	= 79	RLO A
0011	E618	= 80	JNC DIVIE
0013	37	= 81	CPL A
0014	61	= 82	ADD A,@R1
0015	37	= 83	CPL A
0016	0420	= 84	JMP DIVIC
		= 85 :3	DIVIDEND(15-8):=DIVIDEND(15-8)-DIVISOR
0018	37	= 86 DIVIE:	CPL A
0019	61	= 87	ADD A,@R1
001A	37	= 88	CPL A
		= 89 :3	IF BORROW=1 THEN
001B	E620	= 90	JNC DIVIC
		= 91 :4	RESTORE DIVIDEND
001D	61	= 92	ADD A,@R1
001E	0421	= 93	JMP DIVID
		= 94 :3	ELSE
		= 95 DIVIC:	
		= 96 :4	QUOTIENT(0) =1
0020	1A	= 97	INC XA
		= 98 :3	ENDIF
		= 99 :3	COUNT =COUNT-1
		= 100 :2	UNTIL COUNT=0
0021	E80C	= 101 DIVID:	DJNZ COUNT,DIVILP
		= 102 :2	CLEAR OVERFLOW FLAG
0023	97	= 103	CLR C
		= 104 :1	ENDIF
		= 105 :1	ENDDIVIDE
0024	2B	= 106 DIVIS:	NCH A,XA
0025	83	= 107	RET
		108	END

USER SYMBOLS											
COUNT	0003	DIV16	0000	DIV1A	0006	DIV1B	0024	DIVIC	0020	DIVID	0021
DIVIE	0018	DIVILP	000C								

ASSEMBLY COMPLETE, NO ERRORS

Figure 6 (continued)

All mnemonics copyrighted © Intel Corporation 1979.

BINARY AND BCD CONVERSIONS

The conversion of a binary value to a BCD (binary coded decimal) number can be done with a very straightforward algorithm:

```

CONVERT_TO_BCD:
BCDACCUM:=0
COUNT:=PRECISION
REPEAT
    BIN:=BIN * 2
    BCD:=BCD * 2 + CARRY
    COUNT:=COUNT - 1
UNTIL COUNT=0
END CONVERT_TO_BCD
    
```

The variable **BCDACCUM** is a BCD string used to accumulate the result; the variable **BIN** is the binary number to be converted. **PRECISION** is a constant which gives the length, in binary bits of BIN. To see how this works, assume that BIN is a sixteen bit value with the most significant bit set. On the first pass through the loop the multiplication of **BIN** will result in a carry and this carry will be added to BCD. On the remaining passes through the loop BCD will be multiplied by two 15 times. The initial carry into BCD will be multiplied by 2^{15} or 32678, which is the "value" of the most significant bit of **BIN**. The process repeats with each bit of **BIN** being introduced to **BCDACCUM** and then being scaled up on successive passes through the loop. Figure 7 shows the implementation of this algorithm for the 8049.



IS15-11 MCS-48/UPI-41 MACRO ASSEMBLER, V2.0

```

LOC OBJ      SEQ      SOURCE STATEMENT
          1 $MACROFILE
          2 $INCLUDE( F1.CONBCD.HED)
= 3 ;*****
= 4 ;*
= 5 ;*      CONBCD
= 6 ;*
= 7 ;*****
= 8 ;*
= 9 ;*      THIS UTILITY CONVERTS A 16 BIT BINARY VALUE TO BCD
= 10 ;*     AT ENTRY
= 11 ;*      A = LOWER EIGHT BITS OF BINARY VALUE
= 12 ;*      XA= UPPER EIGHT BITS OF BINARY VALUE
= 13 ;*      R0= POINTER TO A PACKED BCD STRING
= 14 ;*
= 15 ;*     AT EXIT
= 16 ;*      A = UNDEFINED
= 17 ;*      XA= UNDEFINED
= 18 ;*      C = SET IF OVERFLOW ELSE CLEARED
= 19 ;*
= 20 ;*****
21 ;
22 ;
23 $INCLUDE( F1.CONBCD.PDL)
= 24 ;1 CONVERT_TO_BCD
= 25 ;1 BCDACC =0
= 26 ;1 COUNT =16
= 27 ;1 REPEAT
= 28 ;2   BIN =BIN*2
= 29 ;2   BCD =BCD*2+CARRY
= 30 ;2   IF CARRY FROM BCDACC GOTO ERROR EXIT
= 31 ;2   COUNT =COUNT-1
= 32 ;1 UNTIL COUNT=0
= 33 ;1 END CONVERT_TO_BCD
24 ;
25 ; EQUATES
26 ; =====
27 ;
0002      28 XA     EQU     K2
0003      29 COUNT EQU     R3
0004      40 ICNT  EQU     K4
          41 ;
0003      42 DIGPR EQU     3
          43 ;
          44 $EJECT
          45 $INCLUDE( F1.CONBCD)
= 46 ;
0005      47 TEMP1 SET     R5
= 48 ;
= 49 ;1 CONVERT_TO_BCD
= 50 ENBCD
= 51 ;1 BCDACC =0
0000 28    = 52      XCH     A,R0
    
```

Figure 7

All mnemonics copyrighted © Intel Corporation 1979.

```

LOC  OBJ      SEQ      SOURCE STATEMENT

0001  A0      = 53      MOV     R1, A
0002  28      = 54      XCH     A, R0
0003  B002    = 55      MOV     ICNT, #DIGPR
0005  B100    = 56  BCDCC0R  MOV     @R1, #000
0007  19      = 57      INC     R1
0008  EC05    = 58      DJNZ   ICNT, BCDCC0R
          = 59 :1  COUNT =16
000A  B810    = 58      MOV     COUNT, #16
          = 61 :1  REPEAT
          = 62  BCDCC0B.
          = 63 :2  BIN:=BIN*2
000C  97      = 64      CLP     C
000D  F7      = 65      RLC     A
000E  2A      = 66      XCH     A, XA
000F  F7      = 67      RLC     A
0010  2A      = 68      XCH     A, XA
          = 69 :2  BCD =BCD*2+CARRY
0011  28      = 70      XCH     A, R0
0012  A9      = 71      MOV     R1, A
0013  28      = 72      XCH     A, R0
0014  B003    = 73      MOV     ICNT, #DIGPR
0016  A0      = 74      MOV     TEMP1, A
0017  F1      = 75  BCD0C  MOV     A, @R1
0018  71      = 76      ADDC   A, @R1
0019  57      = 77      DA     A
001A  A1      = 78      MOV     @R1, A
001B  19      = 79      INC     R1
001C  EC17    = 80      DJNZ   ICNT, BCD0C
001E  F0      = 81      MOV     A, TEMP1
          = 82 :2  IF CARRY FROM BCDACC GOTO ERROR EXIT
001F  F624    = 83      JC     BCDCC0D
          = 84 :2  COUNT =COUNT-1
          = 85 :1  UNTIL COUNT=0
0021  EB0C    = 86      DJNZ   COUNT, BCDCC0B
0022  97      = 87      CLR     C      ; CLEAR CARRY TO INDICATE NORMAL TERMINATION
          = 88 :1  END CONVERT_TO_BCD
0024  83      = 89  BCDCC0D  RET
          = 90  END

USER SYMBOLS
BCDCC0R 0005  BCDCC0B 000C  BCDCC0D 0024  BCD0C 0017  CNBCD 0000  COUNT 0003  DIGPR 0003  ICNT 0004
TEMP1 0005  XA 0002

ASSEMBLY COMPLETE. NO ERRORS

```

Figure 7 (continued)



The conversion of a BCD value to binary is essentially the same process as converting a binary value to BCD.

```

CONVERT_TO_BINARY
  BIN:=0
  COUNT:=DIGNO
  REPEAT
    BCDACCUM:=BCDACCUM*10
    BIN:=10*BIN+CARRY DIGIT
    COUNT:=COUNT-1
  UNTIL COUNT=0
END CONVERT_TO_BINARY
    
```

The only complexity is the two multiplications by ten. The BCDACCUM can be multiplied by ten by shifting it left four places (one digit). The variable BIN could be multiplied using the multiply algorithm already discussed, but it is usually more efficient to do this by mak-

ing the following substitution:

$$BIN = 10 * BIN = (2 * 5) * (BIN) = 2 * (2 * 2 + 1) * BIN$$

This implies that the value 10 * BIN can be generated by saving the value of BIN and then shifting BIN two places left. After this the original value of BIN can be added to the new value of BIN (forming 5 * BIN) and then BIN can be multiplied by two. It is often possible to implement the multiplication of a value by a constant by using such techniques. Figure 8 shows an 8049 routine which converts BCD values to binary. This routine differs slightly from the algorithm above in that the BCD digits are read, and converted to binary, two digits at a time. Protection has also been added to detect BCD operands which, if converted, would yield binary values beyond the range of the result.

IS15-11 MCS-48/UP1-41 MACRO ASSEMBLER, V2.0

```

LOC  OBJ      SEQ      SOURCE STATEMENT
      1 $MACROFILE
      2 $INCLUDE( F1:CONBIN.HED)
      3 :*****
      4 :*
      5 :*      CONBIN
      6 :*
      7 :*****
      8 :*
      9 :*      THIS UTILITY CONVERTS A 6 DIGIT BCD VALUE TO BINARY
     10 :*      AT ENTRY
     11 :*      R0= POINTER TO A PACKED BCD STRING
     12 :*
     13 :*      AT EXIT:
     14 :*      A = LOWER EIGHT BITS OF THE BINARY RESULT
     15 :*      XA= UPPER EIGHT BITS OF THE BINARY RESULT
     16 :*      C = SET IF OVERFLOW ELSE CLEARED
     17 :*
     18 :*****
     19 :
     20 :
     21 $INCLUDE( F1:CONBIN.PDL)
     22 :
     23 :
     24 :1 CONVERT_TO_BINARY
     25 :1 POINTER0:=POINTER0+DIGITPAIR-1
     26 :1 COUNT:=DIGITPAIR
     27 :1 BIN:=0
     28 :1 REPEAT
     29 :2   BIN:=BIN*10
     30 :2   BIN:=BIN+MEM(R0)[7-4]
     31 :2   BIN:=BIN*10
     32 :2   BIN:=BIN+MEM(R0)[3-0]
    
```

All mnemonics copyrighted © Intel Corporation 1979.

LOC	OBJ	SEQ	SOURCE STATEMENT
		= 33 :2	POINTER0:=POINTER0-1
		= 34 :2	COUNT:=COUNT-1
		= 35 :1	UNTIL COUNT=0
		= 36 :1	END CONVERT_TO_BINARY
		37 :	
		38 :	EDUATES
		39 :	=====
		40 :	
0002		41 :R	EDU R2
0003		42 COUNT	EDU R3
0004		43 ICNT	EDU R4
		44 :	
0003		45 DIGPR	EDU 3
		46 :	
		47 :	#EJECT
		48 :	#INCLUDE('F1.CONBIN)
		= 49 :	
0005		= 50 TEMP1	SET R5
0006		= 51 TEMP2	SET R6
		= 52 :	
		= 53 :1	CONVERT_TO_BINARY
		= 54 :	CONBIN:
		= 55 :1	POINTER0:=POINTER0+DIGITPAIR-1
0000 F8		= 56 :	MOV A,R0
0001 0302		= 57 :	ADD A,#DIGPR-1
0003 08		= 58 :	MOV R0,A
		= 59 :1	COUNT:=DIGITPAIR
0004 8B03		= 60 :	MOV COUNT,#DIGPR
		= 61 :1	BIN=0
0006 27		= 62 :	CLR A
0007 0A		= 63 :	MOV :R,A
		= 64 :1	REPEAT
		= 65 :	CONBLP:
		= 66 :2	BIN:=BIN*10
0008 142B		= 67 :	CALL CONB10
000A F62A		= 68 :	JC CONBER
		= 69 :2	BIN:=BIN+MEM(R0)(7-4)
000C 0D		= 70 :	MOV TEMP1,A
000D F0		= 71 :	MOV A,@R0
000E 47		= 72 :	SWAP A
000F 530F		= 73 :	ANL A,#0FH
0011 6D		= 74 :	ADD A,TEMP1
0012 2A		= 75 :	XCH A,:A
0013 1300		= 76 :	ADDC A,#00
0015 2A		= 77 :	XCH A,:A
0016 F62A		= 78 :	JC CONBER
		= 79 :2	BIN:=BIN*10
0018 142B		= 80 :	CALL CONB10
001A F62A		= 81 :	JC CONBER
		= 82 :2	BIN:=BIN+MEM(R0)(3-0)
001C 0D		= 83 :	MOV TEMP1,A
001D F0		= 84 :	MOV A,@R0
001E 530F		= 85 :	ANL A,#0FH
0020 6D		= 86 :	ADD A,TEMP1
0021 2A		= 87 :	XCH A,:A

```

LOC  OBJ      SEQ      SOURCE STATEMENT
0022 1300      = 88      ADDC  A, #00
0024 2A        = 89      XCH  A, XA
0025 F62A      = 90      JC   CONBER
          = 91 ; 2  POINTER0:=POINTER0-1
0027 08        = 92      DEC  R0
          = 93 ; 2  COUNT:=COUNT-1
          = 94 ; 1  UNTIL COUNT=0
0028 EB08      = 95      DJNZ COUNT, CONBLP
002A 83        = 96 ; 1  END CONVERT_TO_BINARY
          = 97 CONBER: RET
          = 98 #EJECT
          = 99 ;
          = 100 ;
          = 101 ;      UTILITY TO MULTIPLY BIN BY 10
          = 102 ;      CARRY WILL BE SET IF OVERFLOW OCCURS
          = 103 ;
002B AD        = 104 CONB10: MOV  TEMP1, A ; SAVE A
002C 2A        = 105      XCH  A, XA ; SAVE XA
002D AE        = 106      MOV  TEMP2, A
002E 2A        = 107      XCH  A, XA
          = 108 ;
002F 97        = 109      CLR  C
0030 F7        = 110      RLC  A ; BIN:=BIN*2
0031 2A        = 111      XCH  A, XA
0032 F7        = 112      RLC  A
0033 2A        = 113      XCH  A, XA
0034 F646      = 114      JC   CONB1E ; ERROR ON OVERFLOW
          = 115 ;
0036 F7        = 116      RLC  A ; BIN:=BIN*4
0037 2A        = 117      XCH  A, XA
0038 F7        = 118      RLC  A
0039 2A        = 119      XCH  A, XA
003A F646      = 120      JC   CONB1E ; ERROR ON OVERFLOW
          = 121 ;
003C 6D        = 122      ADD  A, TEMP1 ; BIN:=BIN*5
003D 2A        = 123      XCH  A, XA
003E 7E        = 124      ADDC A, TEMP2
003F 2A        = 125      XCH  A, XA
0040 F646      = 126      JC   CONB1E ; ERROR ON OVERFLOW
          = 127 ;
0042 F7        = 128      RLC  A ; BIN:=BIN*10
0043 2A        = 129      XCH  A, XA
0044 F7        = 130      RLC  A
0045 2A        = 131      XCH  A, XA
          = 132 ;
0046 83        = 133 CONB1E: RET
          = 134
          = 135 ;
          = 136 END

USER SYMBOLS
CONB10 002B  CONB1E 0046  CONBER 002A  CONBIN 0000  CONBLP 0008  COUNT 0003  DIGPR 0003  ICNT 0004
TEMP1 0005  TEMP2 0006  XA 0002

```

ASSEMBLY COMPLETE, NO ERRORS

All mnemonics copyrighted © Intel Corporation 1979.

CONCLUSION

The design goals of the full duplex serial communications software were realized; if transmission and reception are occurring concurrently, only 42 percent of the real time available to the 8049 will be consumed by the serial link. This implies that an 8049 running full duplex serial I/O will still outperform earlier members of the family running without the serial I/O requirement. It is also possible to run this program in an 8048 or 8748 at 1200 baud with the same 42 percent CPU utilization.

The execution times for the other routines that have been discussed have been summarized in Table 1. All of these routines were written to maintain maximum usability rather than minimum code size or execution time. The resulting execution times and code size are therefore what the user can expect to see in a real application. The results that were obtained clearly show the efficiency and speed of the 8049. The equivalent times for the 8048 are also shown. It is clear that the 8049 represents a substantial performance advantage over the 8048. Considering, in most applications, that the 8048 is

the highest performance microcomputer available to date, the performance advantage of the 8049 should allow the cost benefits of a single chip microcomputer to be realized in many applications which up until now have required too much "computer power" for a single chip approach.

	EXECUTION TIME (MICROSECONDS)		
	BYTES	8049	8048
MPY8	21	109	200
DIV 16	37	183 MIN 204 MAX	335 MIN 375 MAX
CONBCD	36	733	1348
CONBIN	70	388	713

Table 1. Program Performance