8XC196KR Peripherals:
A User’s Point of View

ROB KOWALCZYK
STEVE McINTYRE

April 1992
# 8XC196KR Peripherals: A User's Point of View

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 INTRODUCTION</td>
<td>7</td>
</tr>
<tr>
<td>1.1 8XC196KR Overview</td>
<td>8</td>
</tr>
<tr>
<td>1.1.1 General Description—CPU</td>
<td>8</td>
</tr>
<tr>
<td>1.1.2 Integrated I/O Subsystem</td>
<td>9</td>
</tr>
<tr>
<td>1.2 New 8XC196KR Instructions</td>
<td>9</td>
</tr>
<tr>
<td>1.2.1 52 Lead Device</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Windowing</td>
<td>11</td>
</tr>
<tr>
<td>1.3.1 Examples of Vertical Windows</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Top 5 Issues With Windowing</td>
<td>14</td>
</tr>
<tr>
<td>2.0 INTERRUPTS AND THE PERIPHERAL TRANSACTION SERVER (PTS)</td>
<td>14</td>
</tr>
<tr>
<td>2.1 PTS Execution</td>
<td>15</td>
</tr>
<tr>
<td>2.2 PTS Modes</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Single Transfer Mode</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Single Transfer Mode Example</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3 Block Transfer Mode</td>
<td>17</td>
</tr>
<tr>
<td>2.2.4 Block Transfer Mode Example</td>
<td>19</td>
</tr>
<tr>
<td>2.2.5 A/D Scan Mode, PWM Mode and PWM Toggle Mode</td>
<td>22</td>
</tr>
<tr>
<td>2.3 PTS Latency Times</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Top 5 Issues with PTS</td>
<td>22</td>
</tr>
<tr>
<td>3.0 UNDERSTANDING THE PORTS</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Port 0</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Port 1 / 2 / 6</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Port 3 / 4</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Port 5</td>
<td>26</td>
</tr>
<tr>
<td>3.5 Top 5 Issues with the Ports</td>
<td>27</td>
</tr>
<tr>
<td>4.0 SERIAL I/O PORT (SIO PORT)</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Serial Port SFRs</td>
<td>27</td>
</tr>
<tr>
<td>4.1.1 SP__CONTROL</td>
<td>28</td>
</tr>
<tr>
<td>4.1.2 SP__STATUS</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Baud Rate Generation</td>
<td>28</td>
</tr>
<tr>
<td>4.3 SIO Port Configuration</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Mode 0: Synchronous Communications</td>
<td>30</td>
</tr>
<tr>
<td>4.5 Mode 1: Standard Asynchronous Serial I/O</td>
<td>30</td>
</tr>
<tr>
<td>4.5.1 Setting Up Mode 1 Operation</td>
<td>31</td>
</tr>
<tr>
<td>4.5.2 SIO and the PTS</td>
<td>32</td>
</tr>
<tr>
<td>4.6 Modes 2 and 3: 9 Bit Communications Modes</td>
<td>35</td>
</tr>
<tr>
<td>4.7 Top 5 Issues with the SIO</td>
<td>36</td>
</tr>
<tr>
<td>5.0 SYNCHRONOUS SERIAL I/O AND PERIPHERAL TRANSACTION SERVER</td>
<td>36</td>
</tr>
<tr>
<td>5.1 SSIO Port SFRs</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Example 1</td>
<td>38</td>
</tr>
<tr>
<td>5.3 Using the PTS and Handshake Mode</td>
<td>38</td>
</tr>
<tr>
<td>5.4 SSIO and the PTS</td>
<td>39</td>
</tr>
<tr>
<td>5.5 Top 5 Issues with the SSIO</td>
<td>40</td>
</tr>
<tr>
<td>6.0 ANALOG TO DIGITAL CONVERTER</td>
<td>41</td>
</tr>
<tr>
<td>6.1 A/D Command Register (AD__COMMAND)</td>
<td>41</td>
</tr>
<tr>
<td>6.2 A/D Time Register (AD__TIME)</td>
<td>41</td>
</tr>
<tr>
<td>6.3 A/D Test Register (AD__TEST)</td>
<td>42</td>
</tr>
<tr>
<td>6.4 A/D Result Register (AD__RESULT)</td>
<td>43</td>
</tr>
<tr>
<td>6.5 Example A/D Programs</td>
<td>45</td>
</tr>
<tr>
<td>6.5.1 Using the A/D with the PTS</td>
<td>45</td>
</tr>
<tr>
<td>6.6 Threshold Detection</td>
<td>49</td>
</tr>
<tr>
<td>6.7 A/D Test Modes</td>
<td>49</td>
</tr>
<tr>
<td>6.8 Top 5 Issues with the A/D</td>
<td>50</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>PAGE</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.0 EVENT PROCESSOR ARRAY</td>
<td></td>
</tr>
<tr>
<td>(EPA)</td>
<td>50</td>
</tr>
<tr>
<td>7.1 Timers</td>
<td>50</td>
</tr>
<tr>
<td>7.1.1 Timer Examples</td>
<td>51</td>
</tr>
<tr>
<td>7.2 EPA Input/Output Structure</td>
<td>51</td>
</tr>
<tr>
<td>7.3 EPA Interrupts</td>
<td>53</td>
</tr>
<tr>
<td>7.4 Input Capture</td>
<td>54</td>
</tr>
<tr>
<td>7.4.1 HSI Example #1</td>
<td>54</td>
</tr>
<tr>
<td>7.4.2 HSI Example #2: ABS</td>
<td>55</td>
</tr>
<tr>
<td>7.5 EPA HSO Generation</td>
<td>59</td>
</tr>
<tr>
<td>7.5.1 Square Wave Generation</td>
<td>59</td>
</tr>
<tr>
<td>7.5.2 PWM Signal Generation</td>
<td></td>
</tr>
<tr>
<td>Without PTS</td>
<td>61</td>
</tr>
<tr>
<td>7.5.3 PWM Generation With PTS</td>
<td>63</td>
</tr>
<tr>
<td>7.5.4 PWM Generation Using</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>66</td>
</tr>
<tr>
<td>7.6 Top 5 Issues with the EPA</td>
<td>69</td>
</tr>
</tbody>
</table>
Figures

1-1 Figure 1-1. 8XC196KR Block Diagram .................................. 7
1-2 Figure 1-2. 8XC196KR Memory Map ........................................ 8
1-3 Figure 1-3. Special Function Registers .................................. 8
1-4 Figure 1-4. Special Function Registers .................................. 9
1-5 Figure 1-5. 128-Byte Windows ........................................... 12
1-6 Figure 1-6. 64-Byte Windows ............................................. 12
1-7 Figure 1-7. 32-Byte Windows ............................................. 12
2-1 Figure 2-1. 8XC196KR Interrupt Priorities ................................. 15
2-2 Figure 2-2. PTS Control Blocks (PTSCB) ................................ 16
2-3 Figure 2-3. PTS Control Single Transfer ................................. 16
2-4 Figure 2-4. PTS Control Block Transfer ................................ 19
2-5 Figure 2-5. PTS Interrupt Response Time ................................ 22
3-1 Figure 3-1. Input Port 0 Structure ........................................ 23
3-2 Figure 3-2. Ports 1, 2, 5 & 6 (and 3 / 4 - see notes) .................... 24
3-3 Figure 3-3. Port 1, 2, and 6 Truth Table ................................. 25
3-4 Figure 3-4. Port Reset Values ........................................... 25
3-5 Figure 3-5. Port 5 Truth Table ........................................... 27
4-1 Figure 4-1. SP__CONTROL Register .................................... 27
4-2 Figure 4-2. SP__STATUS Register ...................................... 28
4-3 Figure 4-3. SP__BAUD Register Equations ................................ 29
4-4 Figure 4-4. Common Baud Rate Values ................................ 29
4-5 Figure 4-5. Serial Port Frames, Mode 1, 2 and 3 ........................ 30
5-1 Figure 5-1. SSIO Control Register .................................... 37
5-2 Figure 5-2. SSIO Transmit/Receive Timings ............................... 37
6-1 Figure 6-1. AD__COMMAND Register .................................. 41
6-2 Figure 6-2. AD__TIME Register ........................................ 42
6-3 Figure 6-3. A/D Error vs. Conversion Time .............................. 42
6-4 Figure 6-4. AD__TEST Register ......................................... 42
6-5 Figure 6-5. AD__RESULT Register ...................................... 43
6-6 Figure 6-6. A Typical A/D Transfer Function Error, with Offset and Full Scale Errors ............................... 44
6-7 Figure 6-7. Program Segment to Initialize A/D and Convert on ACH5 .................................. 45
6-8 Figure 6-8. Example A/D Scan Mode Table ............................... 46
7-1 Figure 7-1. TIMER__CONTROL Register ................................ 50
7-2 Figure 7-2. EPA__CONTROL Register .................................. 51
7-3 Figure 7-3. The EPA__PEND and EPA__MASK Registers ............... 53
7-4 Figure 7-4. EPA Interrupt Priority Vector ............................... 53
7-5 Figure 7-5. Wheel Speed Signal for each Wheel ........................ 55
7-6 Figure 7-6. Output Generated by Program 11 ............................ 60
7-7 Figure 7-7. Output of Program 12 and 13 ............................... 63
Programs

Program 1a, b. Send 30 bytes over the SIO using the PTS in Single Xfer Mode .......... 18, 19
Program 2a, b. Using the EXTINT with the PTS Block Transfer Mode ............... 20, 21
Program 3. SIO Communication via Polling the SP__Status Bits (TI and RI) .......... 31
Program 4a, b, c. Using the PTS with both the TI and RI Interrupts ............... 32, 33, 34
Program 5. SSIO, Send One Byte ................................................. 38
Program 6. SSIO, Send One Byte in Handshake Mode ................................ 39
Program 7. SSIO and the PTS ....................................................... 40
Program 8a, b. A/D Scan Mode using the PTS ..................................... 47, 48
Program 9. Start an A/D Conversion on a Positive Input Edge ...................... 54
Program 10a, b, c. ABS Input Frequency Detection using the PTS and EPA Inputs ... 56, 57, 58
Program 11. Generating 2 PWM Pulses Using No CPU Overhead .................. 60
Program 12. PWM Generation Using Interrupts .................................... 62
Program 13. Generate a PWM on EPA0 using the PTS Toggle Mode ................ 64
Program 14. Generate a PWM Using the PTS PWM Mode and Re-Map Feature ...... 65
Program 15a, b. Generate a PWM Output Using EPA9 and Software Interrupts .... 67, 68
1.0 INTRODUCTION

High Speed Event control is a common occurrence in today's control applications. Also mixing analog and digital control in the same application is becoming a necessity.

In 1982 Intel introduced the first member of the 16-bit microcontroller family (MCS-96): the 8096 device. This family has grown from that first introduction to today's 4th generation of highly integrated, 1 micron CHMOS technology members. The 8XC196KR, 8XC196KQ, 8XC196JR, and 8XC196JQ. (Known hereafter as 8XC196KR).

These devices combine high speed 16- and 32-bit precision calculation capability (100% instruction set compatible with the MCS-96 product family) with a dedicated I/O subsystem that has no equal. Figure 1-1 illustrates the complete functional blocks that make up the 8XC196KR devices.

This Ap-note will briefly describe the 8XC196KR CPU and peripherals with example applications for each.

Figure 1-1. 8XC196KR Block Diagram
1.1 8XC196KR Overview

1.1.1 GENERAL DESCRIPTION – CPU

The 8XC196KR instructions are a true instruction super set of past 8096 devices (8X9XBH and 8XC196KB). It uses a 16-bit ALU which operates on 512 bytes of registers instead of an accumulator. Like the 8096, any location within the 512 byte register file can be used as source or destinations for most of the instruction addressing modes.

This register to register architecture is common to the MCS-96 family. Many of the instructions can operate on bytes, words and double words from anywhere in the 64K byte address space. To assist in the understanding of the 8XC196KR memory, a map is shown in Figure 1-2.

<table>
<thead>
<tr>
<th>Address</th>
<th>Type of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0FFFFh</td>
<td>External Memory</td>
</tr>
<tr>
<td>06000h</td>
<td></td>
</tr>
<tr>
<td>05FFFh</td>
<td>Internal/External EPROM</td>
</tr>
<tr>
<td>02080h</td>
<td></td>
</tr>
<tr>
<td>0207Fh</td>
<td>Internal/External EPROM</td>
</tr>
<tr>
<td>0205Eh</td>
<td>(Int. Vectors/Open/Reserved)</td>
</tr>
<tr>
<td>0205Dh</td>
<td></td>
</tr>
<tr>
<td>02030h</td>
<td>Internal/External EPROM</td>
</tr>
<tr>
<td>0202Fh</td>
<td>(Int. Vectors)</td>
</tr>
<tr>
<td>02020h</td>
<td></td>
</tr>
<tr>
<td>0201Fh</td>
<td>Internal/External EPROM</td>
</tr>
<tr>
<td>02014h</td>
<td>(CCBO/CCB1/Reserved)</td>
</tr>
<tr>
<td>02013h</td>
<td></td>
</tr>
<tr>
<td>02000h</td>
<td>Internal SFR Registers</td>
</tr>
<tr>
<td>01FFFh</td>
<td></td>
</tr>
<tr>
<td>01FF0h</td>
<td></td>
</tr>
<tr>
<td>01EFGh</td>
<td>External Memory</td>
</tr>
<tr>
<td>00500h</td>
<td></td>
</tr>
<tr>
<td>004FFh</td>
<td>Internal Code RAM</td>
</tr>
<tr>
<td>00400h</td>
<td></td>
</tr>
<tr>
<td>003FFh</td>
<td>External Memory</td>
</tr>
<tr>
<td>00200h</td>
<td></td>
</tr>
<tr>
<td>001FFh</td>
<td>Internal Register RAM</td>
</tr>
<tr>
<td>00018h</td>
<td></td>
</tr>
<tr>
<td>00017h</td>
<td></td>
</tr>
<tr>
<td>00000h</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1-2. 8XC196KR Memory Map

The lower 24 bytes of the register file contain Special Function Registers (SFRs) that are used to control on-chip peripherals (similar to past 8096 devices). In addition to these SFRs, the 8XC196KR device has 256 more SFRs located from 1F00H through 1FFFFH. All RAM memory (Register memory from 0000H to 01FFH and Code RAM memory from 0400H to 04FFH) can be kept alive using the low current power-down or idle modes.

Accessing RESERVED SFR/data memory locations are not allowed. Memory locations 1F00h through 1F5Fh are also considered RESERVED.

Figures 1-3 and 1-4 show the layout and reset values of the SFRs on the 8XC196KR. Most of these registers are Read and Writable (unlike those in past 8096 devices).

<table>
<thead>
<tr>
<th>SFR</th>
<th>Location</th>
<th>RESET Value</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPIN</td>
<td>1FDAH</td>
<td>XXH</td>
<td>R</td>
</tr>
<tr>
<td>P3PIN</td>
<td>1FFEH</td>
<td>XXH</td>
<td>R</td>
</tr>
<tr>
<td>P3REG</td>
<td>1FFCH</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P4PIN</td>
<td>1FFFH</td>
<td>XXH</td>
<td>R</td>
</tr>
<tr>
<td>P4REG</td>
<td>1FFDH</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P1PIN</td>
<td>1FD6H</td>
<td>XXH</td>
<td>R</td>
</tr>
<tr>
<td>P1REG</td>
<td>1FD4H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P1IO</td>
<td>1FD2H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P1SSEL</td>
<td>1FD0H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>P2PIN</td>
<td>1FCFH</td>
<td>1XXXXXXXB</td>
<td>R</td>
</tr>
<tr>
<td>P2REG</td>
<td>1FCDH</td>
<td>7FH</td>
<td>R/W</td>
</tr>
<tr>
<td>P2IO</td>
<td>1FCBH</td>
<td>7FH</td>
<td>R/W</td>
</tr>
<tr>
<td>P2SSEL</td>
<td>1FC9H</td>
<td>80H</td>
<td>R/W</td>
</tr>
<tr>
<td>P6PIN</td>
<td>1FD7H</td>
<td>XXH</td>
<td>R</td>
</tr>
<tr>
<td>P6REG</td>
<td>1FD5H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P6IO</td>
<td>1FD3H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P6SSEL</td>
<td>1FD1H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>P5PIN</td>
<td>1FF7H</td>
<td>1XXXXXXXB</td>
<td>R</td>
</tr>
<tr>
<td>P5REG</td>
<td>1FF5H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P5IO</td>
<td>1FF3H</td>
<td>0FFH</td>
<td>R/W</td>
</tr>
<tr>
<td>P5SSEL</td>
<td>1FF1H</td>
<td>80H</td>
<td>R/W</td>
</tr>
<tr>
<td>INT__MASK</td>
<td>0008H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>INT__MASK1</td>
<td>0013H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>INT__PEND</td>
<td>0009H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>INT__PEND1</td>
<td>0012H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>PTS____SRV</td>
<td>0006H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>PTS____SELECT</td>
<td>0004H</td>
<td>00H</td>
<td>R/W</td>
</tr>
<tr>
<td>WDT</td>
<td>000AH</td>
<td>00H</td>
<td>R</td>
</tr>
</tbody>
</table>

Figure 1-3. Special Function Registers
1.1.2 INTEGRATED I/O SUBSYSTEM

Some of the I/O features on the 8XC196KR are similar to past 8096 devices. But, a great deal of the I/O and its specific functions have changed for the better.

For example, the WatchDog Timer (WDT) is an internal timer which can be used to reset the system when software fails to operate properly. On past 8096 devices this feature was turned off until initially written. On the 8XC196KR devices, the Chip Configuration Byte (CCB1) contains a bit (bit 3) which can have this feature always enabled. Now if software fails before it gets to the WDT initialization code, it will reset the system.

The 8XC196KR device still contains an Analog to Digital converter, High Speed Input Capture and Output Compare called Event Processor Array (EPA), an integrated 16-bit timer/counter subsystem, and Asynchronous/Synchronous Serial I/O.

In addition to the above peripherals, the 8XC196KR device has an additional Synchronous Serial I/O port, an Additional timer/counter, faster interrupt response capability through the Peripheral Transaction Server (PTS), an 8 bit slave port that allows other CPUs in the system to request information from the 8XC196KR through interrupt control, and an additional 8 pins of I/O.

This integration of I/O, memory, ALU capability, and overall system speed makes the 8XC196KR device a perfect fit in such applications as: Motor Control, Engine Control, Anti-lock Brakes, Suspension Control, Hard Disk Drive Controllers, Printer Control, as well as many others.

1.2 New 8XC196KR Instructions

The 8XC196KR device is an instruction super set of the past 8096 devices (8X9XBH, 8XC196KB). The software used to Assemble / Compile / Link and Locate programs still holds true for the 8XC196KR. (ASM96, RL96, iC96, PL/M96, ....)

The 8XC196KR device has additional instructions not seen before by MCS-96 programmers. These 6 new instructions are EPTS (Enable PTS) / DPTS (Disable PTS), XCH (eXCHange word) / XCHB (eXCHange Byte), BMOVI (Interruptable Block MOVe), and TIJMP (Table Indirect JuMP).
Interruptable Block Move has the same form and function as the BMOV instruction except the interrupt request status is checked after each move is completed. If an interrupt is pending and unmasked the block move operation is suspended and the interrupt service routine is invoked. Following the end of the interrupt service routine the block move continues. The BMOVI instruction, unlike the BMOV, will update the counter register, if interrupted.

Assembly Language:

\[
\text{BMOVI } \text{PTRs} \quad \text{CNT} \\
\quad \text{Lreg} , \quad \text{Wreg}
\]

Object code:

\[
0CDH \quad \text{<wreg> } \quad \text{<lreg>}
\]

Enable PTS (Peripheral Transaction Server) by clearing the PSE flag in the PSW register.

Assembly Language:

\[
\text{DPTS}
\]

Object code:

\[
0ECH
\]

Enable PTS (Peripheral Transaction Server) by setting the PSE flag in the PSW register.

Assembly Language:

\[
\text{EPTS}
\]

Object code:

\[
0EDH
\]

Table Indirect Jump, jumps to an address selected out of a table of addresses. This is a three operand instruction with one operand pointing to the base of the jump table, a second pointing indirectly to a 7-bit index value (0 to 128 decimal) and a third is an immediate operand (7-bit) which is used as a mask for the index value.

Assembly Language:

\[
\text{TIJMP} \quad \text{BASE INDEX} \quad \text{MASK} \\
\quad \text{Wreg1,}[Wreg2], \quad \#\text{Mask}
\]

Object code:

\[
0E2H \quad \text{<wreg2> } \quad \text{<mask> } \quad \text{<wreg1>}
\]

XCH/XCHB Exchange Word and Exchange Byte, exchanges the contents of two memory locations. The immediate and indirect addressing modes are NOT supported, only the direct and indexed (short and long).

Assembly Language:

\[
\begin{align*}
\text{DST} & & \text{SRC} \\
\text{XCHB} & & \text{Breg1, Breg2} \\
\text{XCHB} & & \text{Breg1, Offset [Wreg2]} \\
\text{XCH} & & \text{Wreg1, Wreg2} \\
\text{XCH} & & \text{Wreg1, Offset [Wreg2]}
\end{align*}
\]

Object Code:

\[
\begin{align*}
14H & \quad \text{Breg2} & \quad \text{Breg1} \\
1BH & \quad \text{Wreg2, offset_low | offset_high} & \quad \text{breg1} \\
04H & \quad \text{Wreg2} & \quad \text{Wreg1} \\
0BH & \quad \text{Wreg2, offset_low | offset_high} & \quad \text{wreg1}
\end{align*}
\]

For short indexed addressing modes the second offset byte is omitted from the object code stream. For long indexed addressing modes, both bytes of offset are required, making the instruction a 5 byte instruction.

1.2.1 52 LEAD DEVICES

Intel offers a 52 lead version of the 8XC196KR device: the 8XC196JR and 8XC196JQ devices. The first samples and production units use the 8XC196KR die and bond it out in a 52 lead package.

It is important to point out some functionality differences because of future devices or to remain software consistent with the 68 lead device. Because of the absence of pins on the 52 lead device some functions are not supported.

52 Lead Unsupported Functions:

- Analog Channels 0 and 1.
- INST pin functionality.
- SLPINT pin support.
- HLD# or HLDA# functionality.
- External clocking/direction of Timer1.
- WRH# or BHE functions.
- Dynamic buswidth.
- Dynamic wait state control.

The following is a list of recommended practices when using the 52 lead device:

1. **External Memory.** Use an 8-bit bus mode only. There is neither a WRH# or BUSWIDTH pin. The bus can not dynamically switch from 8- to 16-bit or vice versa. Set the CCB bytes to an 8-bit only mode, using WR# function only.
(2) **Wait State Control.** Use the CCB bytes to configure the maximum number of wait states. If the READY pin is selected to be a system function, the device will lockup waiting for READY. If the READY pin is configured as LSIO (default after RESET), the internal logic will receive a logic “0” level and insert the CCB defined number of wait states in the bus cycle. **DON’T USE IRC “111”.**

(3) **NMI support.** The NMI is not bonded out. Make the NMI vector at location 203Eh vector to a Return instruction. This is for glitch safety protection only.

(4) **Auto-Programming Mode.** The 52 lead device will ONLY support the 16-bit zero wait state bus during auto-programming.

(5) **EPA4 through EPA7.** Since the JR and JQ devices use the KR silicon, these functions are in the device, just not bonded out. A programmer can use these as compare only channels or for other functions like software timer, start and A/D, or reset timers.

(6) **Slave Port Support.** The Slave port can still be used on the 52 lead devices. The only function removed is the SLPINT output function.

(7) **Port Functions.** Some port pins have been removed. P5.7, P5.6, P5.5, P5.1, P6.2, P6.3, P1.4 through P1.7, P2.3, P2.5, P0.0 and P0.1. The PxREG, PxSSEL, and PxIO registers can still be updated and read. The programmer should not use the corresponding bits associated with the removed port pins to conditionally branch in software. Treat these bits as RESERVED.

Additionally, these port pins should be setup internally by software as follow:
1. Written to PxREG as “1” or “0”.
2. Configured as Push/Pull, PxIO as “0”.
3. Configured as LSIO.

This configuration will effectively strap the pin either high or low. **DO NOT Configure as Open Drain output “1”; or as an Input pin. This device is CMOS.**

### 1.3 Windowing

The 8XC196KR contains 512 bytes of memory, located from 00h to 1FFh. An additional 256 bytes of on chip SFRs (Special Function Registers) located at 1F00h–1FFFFh. Accesses directly to any location other than 00h–FFh would require a 16 bit address.

The 8XC196KR device has a mechanism known as vertical windowing which allows portions of 16-bit memory (0000–1FFh and 1F00h–1FFFFh) to be remapped to an 8-bit address in the 0080h - 00FFh register RAM area.

Any address accesses using an 8-bit re-mapped address will be windowed through to the 16-bit address.

The 8XC196KR core has the capability to add up to 1K of register RAM (0000–03FFh) and 1K of SFR space (1C00–1FFFFh). However, only 312 bytes of register RAM is accessible along with 256 bytes of SFRs (locations 1F00–1FFFFh). 1F00h through 1FFFFh is considered RESERVED. Only the non-RESERVED locations should be selected for windowing. Any attempt to window outside this area will result in reading of all 1’s and writing to the bit bucket. In addition, the SFRs located from 1FE0h–1FFFFh can NOT be accessed through any window. Any attempt to write these register through a window will have no effect on these SFRs; reading these registers through a window will result in FFh or FFFFh being read.

Windows can be selected to be either 32, 64 or 128 bytes, and will be mapped into locations E0h - F8h, C0h–FFh, or 80h–FFh, respectively. Control over the window is obtained through the use of the WSR register located at 14h. The bit map of the WSR depends somewhat on the size of the window. The MSB of the WSR is not used for windowing, but rather is used to control whether or not outside bus masters can request control of the external bus (HOLD/HLDA enabling).

The next three bits either determine the size of the window, or, for 64 and 32 byte windows, part of the offset address.

Bits 6, 5 and 4 determine which “window” size from memory is to be used. Bits 3, 2, 1, and 0 determine which block the device will window to.

For example, suppose the programmer wanted a window of 128 bytes. First, visualize the memory as being divided into consecutive blocks of 128 bytes each. Note that there is a gap from 400h–1EFFh. This will make 15 blocks of 128 bytes that can be windowed through 80h–0FFh (skipping memory between 400h–1EFFh).

Number the blocks starting with zero. Furthermore, assume that the programmer wanted to window addresses from 180h–1FFh to 80h–FFh. This is the third block of 128 bytes in memory. The fifteen block will be 1F80h–1FFFFh (starting with 0000h). This means that the WSR should contain “001” in the upper bits 6, 5, and 4, to select 128 byte windowing, and “0011” (3) in the lower nibble to select block number 3.
Figures 1-5, 1-6, and 1-7 illustrate all valid codes for the WSR (on the 8XC196KR), and the corresponding windows opened. Using other WSR values (other than "00") is not supported or recommended.

The WSR register is stacked on a PUSHA instruction. This will save the WSR value on the stack while executing an Interrupt Service Routine (ISR). A PUSHA instruction has no affect on the WSR register. Before returning from the ISR, a POPA will return the WSR to the previous value prior to entering the ISR.
1.3.1 EXAMPLES OF VERTICAL WINDOWS

To fully understand the windowing capability, some working examples are needed. Window 1Fh will be used; this remaps memory from 1F80h–1FFFh to 0080h–00FFh (00E0–00FFh have no effect on the SFRs because they will be windowed to memory mapped I/O locations 1F00h through 1FFFh).

Assume the following code segment is executed without windowing. The results of this will be compared with the same code being executed with a window active.

(1) ClrB WSR
(2) Ld 9Ah, #3000h
(3) Ld 86h, 9Ah
(4) Ld 70h, 86h[0]
(5) Ld 72h, 1F9Ah[0]
(6) Ld 82h, [9Ah]
(7) St 76h, 0Ah[9Ah]

Three assumptions shall be made concerning the state of memory before the above code is executed: 1) Location 3000h contains 0303h, 2) location 76h contains 0A0Ah, and 3) the value in timer1 is 1111h (timer is not active). After the code is executed, the registers will be in the following state:

(1) WSR: 00h
(2) 9Ah: 3000h
(3) 86h: 3000h
(4) 70h: 3000h
(5) 72h: 1111h
(6) 82h: 0303h
(7) 300Ah: 0A0Ah

These results are consistent with what would normally be expected without knowledge of windowing. Let the following code be executed with all of the above assumptions intact, but with windowing. Locations from 1F80h–1FFFh will be windowed through 0080h–00FFh.

(1) LdB WSR, #1Fh
(2) Ld 9Ah, #1234h
(3) Ld 86h, 9Ah
(4) Ld 70h, 86h[0]
(5) Ld 72h, 1F9Ah[0]
(6) Ld 82h, [9Ah]
(7) St 76h, 0Ah[9Ah]

Assume that location 1234h contains 0202h, 0080h contains 3000h, and 76h still contains 0A0Ah. The following results will be obtained:

(1) WSR: 1Fh
(2) 1F9Ah: 1234h
(3) 1F86h: 1234h
(4) 70h: 3000h
(5) 72h: 1234h
(6) 1F82h: 0202h
(7) 123Eh: 0A0Ah

Contrast this with the results from the first code segment and note the following differences.

(1) The WSR is loaded with the windowed value that remaps 80h through 0FFh to 1F80h through 1FFFh.
(2) The immediate addressing mode moves immediate data through the window, into the 16-bit address of 1F9Ah, not 009Ah.
(3) Using the direct addressing mode, both the source (9Ah) and the destination (86h) are affected by the open window. This will move data from absolute address 1F9Ah (TIMER1) and place it in absolute address 1F86h (EPA_TIME8).
(4) Here the short indexed addressing mode is used to load register 70h from absolute location 80h + 00h. Notice that windowing does not affect any part of this example. Location 70h is not in the window. The indexed offset value (80h) is a constant and is NOT 1F80h. And lastly, register 0 (00) is not windowable.
(5) This is similar to example (4). Only the long indexed addressing mode is used. Here the 16-bit offset (1F9Ah) is added to the contents of the 00 register to get the address 1F9Ah. The contents of 1F9Ah is then stored in register 72h. No window affect.
(6) The indirect example has a great affect by windowing. Both the source and destination for this addressing mode refers to 8-bit registers. Hence, the contents of 1F9Ah is read and used as a pointer to a 16-bit address. That value is stored through the window to 1F82h (EPA_TIME8). If Auto-incrementing were used, the register 9Ah (1F9Ah through the window) would be incremented by 2.
(7) Lastly the Indexing mode example with the index register being affected by the window. Register 9Ah (1F9Ah through the window) is read and offset by immediate value #0Ah forming 123Eh. This address (123Eh) is the destination for register 76h (unaffected by the window).

As a final example, consider the following piece of code; try to determine exactly what it does before reading on.

(1) LdB WSR, #40h
(2) OrB 0F3h, #18h
(3) LdB 80h, 9Ah
(4) LdB 0EAh, #1Eh
(5) StB 80h, 0Eh
(6) St 0E0h, 0F8h
(7) LdB 0F4h, 0E0h

Contrast this with the results from the first code segment and note the following differences.

(1) The WSR is loaded with the windowed value that remaps 80h through 0FFh to 1F80h through 1FFFh.
(2) The immediate addressing mode moves immediate data through the window, into the 16-bit address of 1F9Ah, not 009Ah.
(3) Using the direct addressing mode, both the source (9Ah) and the destination (86h) are affected by the open window. This will move data from absolute address 1F9Ah (TIMER1) and place it in absolute address 1F86h (EPA_TIME8).
(4) Here the short indexed addressing mode is used to load register 70h from absolute location 80h + 00h. Notice that windowing does not affect any part of this example. Location 70h is not in the window. The indexed offset value (80h) is a constant and is NOT 1F80h. And lastly, register 0 (00) is not windowable.
(5) This is similar to example (4). Only the long indexed addressing mode is used. Here the 16-bit offset (1F9Ah) is added to the contents of the 00 register to get the address 1F9Ah. The contents of 1F9Ah is then stored in register 72h. No window affect.
(6) The indirect example has a great affect by windowing. Both the source and destination for this addressing mode refers to 8-bit registers. Hence, the contents of 1F9Ah is read and used as a pointer to a 16-bit address. That value is stored through the window to 1F82h (EPA_TIME8). If Auto-incrementing were used, the register 9Ah (1F9Ah through the window) would be incremented by 2.
(7) Lastly the Indexing mode example with the index register being affected by the window. Register 9Ah (1F9Ah through the window) is read and offset by immediate value #0Ah forming 123Eh. This address (123Eh) is the destination for register 76h (unaffected by the window).
Which window was opened? The WSR was written a 0100000b, which selects a 32 byte window, starting at block 0. So 00h–1Fh was windowed to E0h–FFh. Next, INT__MASK1 was accessed through the window, and the Receive Interrupt (RI) and the Transmit Interrupt (TI) of the serial port were enabled.

The next three instructions are dangerous as they enable the watchdog timer.

The stack pointer is then set to 0000h and finally, the window is reset by accessing the window select register (WSR) through a window!

1.4 Top 5 Issues With Windowing
1. The PUSHA will NOT clear the WSR. But a POPA will restore the WSR from the stack.
2. Both source and destination bytes of the Direct Addressing Mode can be affected by the window selected.
3. The Offset in the Indexed Addressing Mode is NOT windowable.
4. The PTS uses all 16-bit addresses and is therefore unaffected by the window selected.
5. Without proper understanding of the window mechanism, the user can get into big trouble using windows. Beware the window!

2.0 INTERRUPTS AND THE PERIPHERAL TRANSACTION SERVER (PTS)

The PTS is a microcoded hardware interrupt service routine that “steals” bus cycles to execute. It is able to do a special encoded interrupt service routine in the time it takes to execute one instruction.

In a way, the PTS is a one instruction interrupt service routine that executes without stack or PSW being modified, and with minimum CPU overhead. A simple PTS cycle is only 1.875μS at 16 MHz (less than it takes to do a divide and just a 1 state time longer than a multiply).

There is a new bit in the PSW to control the global enable of PTS interrupts (PSE bit in the PSW). This bit is set using the EPTS instruction and disabled using the DPTS instruction which is discussed in the new instruction section of this document.

On past 8096 devices an interrupt requests sets the INT__PEND bit in the core, the core looks at the corresponding INT__MASK bit to see if it should “vector” to a software interrupt service routine. Also provided that the interrupts are enabled (I bit in the PSW). This type of interrupt response is still on the 8XC196KR devices.

The 8XC196KR core implements a small twist to those events. Before “vectoring” to the software interrupt routine, the core checks another bit: the corresponding PTS__SELECT bit (location 04H:WORD). If this bit is also set, the core will vector to a microcoded interrupt service routine INSTEAD of the software interrupt service routine (provided the PSE bit in the PSW is also set).

The PTS is able to perform a DMA-like response to any interrupt source. Figure 2-1 illustrates the location, priority and source for all of the PTS interrupt vectors available on the 8XC196KR.
All PTS interrupts have higher priority than normal software interrupts. There is a vector for the EPAINTx interrupts, but it is NOT possible to do PTS cycles using this vector due to the nature of the INTPEND bit (See the EPA section for details).

There is no PTS vector for the NMI, TRAP, or Unimplemented Opcode. Also the PTS will not function while in Idle.

### 2.1 PTS Execution

As in normal software interrupt response, the current instruction is completed before the PTS interrupt cycle executes. The internal priority handler, handles the requests based on their priority. Next the PTS vector is read from the vector table to get the address of the PTS Control Block (PTSCB).

As with any instruction there are opcodes and operands. The PTSCB is no exception. It defines the "instruction" to execute when an interrupt request comes in. The first word is the "opcode". In some PTS modes the whole word is used, and in others it only uses the high byte of the word. (see individual PTS modes for details). The next three words are the operands of the PTS interrupt cycle.

If the 8XC196KR device is running from external memory, this interrupt vector fetch may be the only evidence that a PTS cycle has executed.

The CPU executes the proper PTS instruction based on the contents of the PTS Control Block (Moving data from one location to another, internal or external - doing the special PWM / PWM toggle - or A/D scan modes).

Instead the PTS executes a single microcoded instruction that resides in Register RAM (locations 0000H to 01FFH). Code RAM can not be used for PTS control blocks.

The PTSCB is a set of registers that defines how the PTS cycle is to be performed.

This PTS control block or RAM registers are always on QUAD word boundaries (address = 00000000xxxxxx0000). The assembler will not give an error message if this QUAD word boundary rule is violated. If the PTS vector points to a non-QUAD word boundary, upon execution of this PTS cycle, the CPU will round down to the nearest QUAD word boundary.

<table>
<thead>
<tr>
<th>Number</th>
<th>Source</th>
<th>Vector Location</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTS15</td>
<td>NMI - RESERVED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT15</td>
<td>Non Maskable Interrupt</td>
<td>203EH</td>
<td>30</td>
</tr>
<tr>
<td>PTS14</td>
<td>EXTINT</td>
<td>205CH</td>
<td>29</td>
</tr>
<tr>
<td>PTS13</td>
<td>reserved</td>
<td>205AH</td>
<td>28</td>
</tr>
<tr>
<td>PTS12</td>
<td>RI</td>
<td>2058H</td>
<td>27</td>
</tr>
<tr>
<td>PTS11</td>
<td>TI</td>
<td>2056H</td>
<td>26</td>
</tr>
<tr>
<td>PTS10</td>
<td>XFR1</td>
<td>2054H</td>
<td>25</td>
</tr>
<tr>
<td>PTS09</td>
<td>XFR0</td>
<td>2052H</td>
<td>24</td>
</tr>
<tr>
<td>PTS08</td>
<td>CBF</td>
<td>2050H</td>
<td>23</td>
</tr>
<tr>
<td>PTS07</td>
<td>IBF</td>
<td>204EH</td>
<td>22</td>
</tr>
<tr>
<td>PTS06</td>
<td>OBE</td>
<td>204CH</td>
<td>21</td>
</tr>
<tr>
<td>PTS05</td>
<td>A/D Done</td>
<td>204AH</td>
<td>20</td>
</tr>
<tr>
<td>PTS04</td>
<td>EPAINT0</td>
<td>2048H</td>
<td>19</td>
</tr>
<tr>
<td>PTS03</td>
<td>EPAINT1</td>
<td>2046H</td>
<td>18</td>
</tr>
<tr>
<td>PTS02</td>
<td>EPAINT2</td>
<td>2044H</td>
<td>17</td>
</tr>
<tr>
<td>PTS01</td>
<td>EPAINT3</td>
<td>2042H</td>
<td>16</td>
</tr>
<tr>
<td>PTS00</td>
<td>EPAINTX (RESERVED)</td>
<td>2040H</td>
<td>15</td>
</tr>
<tr>
<td>INT14</td>
<td>EXTINT Pin</td>
<td>203CH</td>
<td>14</td>
</tr>
<tr>
<td>INT13</td>
<td>RESERVED</td>
<td>203AH</td>
<td>13</td>
</tr>
<tr>
<td>INT12</td>
<td>Receive SIO Interrupt</td>
<td>2038H</td>
<td>12</td>
</tr>
<tr>
<td>INT11</td>
<td>Transmit SIO Interrupt</td>
<td>2036H</td>
<td>11</td>
</tr>
<tr>
<td>INT10</td>
<td>SSIO channel 1 transfer</td>
<td>2034H</td>
<td>10</td>
</tr>
<tr>
<td>INT09</td>
<td>SSIO channel 0 transfer</td>
<td>2032H</td>
<td>9</td>
</tr>
<tr>
<td>INT08</td>
<td>Command Buffer Full SLP</td>
<td>2030H</td>
<td>8</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>Illegal Opcode</td>
<td>2012H</td>
<td>N/A</td>
</tr>
<tr>
<td>SPECIAL</td>
<td>TRAP instruction</td>
<td>2010H</td>
<td>N/A</td>
</tr>
<tr>
<td>INT07</td>
<td>Input Buffer Full</td>
<td>200EH</td>
<td>7</td>
</tr>
<tr>
<td>INT06</td>
<td>Output Buffer Empty</td>
<td>200CH</td>
<td>6</td>
</tr>
<tr>
<td>INT05</td>
<td>A/D Complete</td>
<td>200AH</td>
<td>5</td>
</tr>
<tr>
<td>INT04</td>
<td>EPAINT0</td>
<td>2008H</td>
<td>4</td>
</tr>
<tr>
<td>INT03</td>
<td>EPAINT1</td>
<td>2006H</td>
<td>3</td>
</tr>
<tr>
<td>INT02</td>
<td>EPAINT2</td>
<td>2004H</td>
<td>2</td>
</tr>
<tr>
<td>INT01</td>
<td>EPAINT3</td>
<td>2002H</td>
<td>1</td>
</tr>
<tr>
<td>INT00</td>
<td>EPAINTX</td>
<td>2000H</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2-1. 8XC196KR Interrupt Priorities
Figure 2-2 shows the 5 PTS modes available on the 8XC196KR devices. The bytes in the PTSCB labeled “UNUSED” can be used by the users' program as register RAM space. The PTS does not require information from these UNUSED locations and the data in the UNUSED locations will not be altered in any way.

### 2.2 PTS Modes

The PTSCB defines the mode or type of PTS cycle to perform when the interrupt request comes in. Five modes are provided on the 8XC196KR: a Single Transfer Mode, a Block Transfer Mode, an A/D Scan Mode, and two PWM Modes.

Any of these modes can be used for ANY interrupt source associated with the PTS vectors (except EPAINTx). ie: The A/D Scan was specifically designed to function with the A/D peripheral, but if the user can think of an application where the A/D scan mode would be used with the SIO peripheral, or any other peripheral it can be done.

#### 2.2.1 SINGLE TRANSFER MODE

The Single Transfer Mode of PTS cycle can transfer data from any address to any other address in memory. The data can be a word or a byte, and the source and/or destination pointers can be optionally incremented.

This PTS mode uses six bytes out of the eight byte in the PTS Control Block. The other 2 bytes can be used as regular scratch pad register. The PTS cycle has NO EFFECT on these two registers.

Below is the PTS Control Byte description for the Single Transfer Mode.

<table>
<thead>
<tr>
<th>PTSVEC →</th>
<th>Single Transfer</th>
<th>Block Transfer</th>
<th>A/D Mode</th>
<th>PWM Mode</th>
<th>PWM Toggle</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTS_COUNT</td>
<td>PTS_COUNT</td>
<td>PTS_COUNT</td>
<td>PTS_COUNT</td>
<td>Unused</td>
<td>Unused</td>
</tr>
</tbody>
</table>

**Figure 2-3. PTS Control Single Transfer**

The PTS Vector points to the first byte in the block. That byte is the COUNT register. The COUNT (PTS_COUNT) holds the number of PTS cycles to be performed before a normal software interrupt is called.

The PTS_COUNT register contains a value that is decremented at the end of the PTS cycle. When equal to zero the PTS_SELECT bit is cleared and the PTS_SRV bit is set. This causes a normal software interrupt routine to execute following the last PTS cycle.

The next byte (PTS Vector + 1) contains the PTS Control (PTS_CONTROL). This byte is present for all modes of operation. It defines the PTS mode, whether the PTS cycle is a word or byte transfer, and if the source and/or destination pointers should be incremented and/or updated during and at the end of the PTS cycle.

Figure 2-3 is an illustration of the PTS_CONTROL register for the Single Transfer Mode.
Notice that there is a bit for UPDATE and a bit for INCREMENT. Typically these are used in the Block Transfer Mode. The source and/or destination pointer is incremented if the bit is set. This increment happens after the PTS transfers the single byte or word to the destination pointer address.

The UPDATE bit in the control byte is set if the newly formed address (created by the increment function) is to be placed in the PTS control block source and destination pointer words.

In the Single transfer mode it makes no sense to increment without updating the pointers, or vice versa.

The Next word in the PTSCB is a Source Pointer. This word points to a 16-bit address (anywhere in memory). It can be an SFR location, or point to off chip memory.

If the PTS cycle directs data moves to read or write to external memory, an external bus cycle will be evident externally. If the PTS cycle is directed internally, no external evidence, other than the PTS vector fetch.

The last used word in the PTSCB is the Destination Pointer. It too points to ANYWHERE in the 64K address space.

### 2.2.2 SINGLE TRANSFER MODE EXAMPLE

Suppose that there is a 30 character message that needs to be transmitted out the Serial I/O port. The PTS can be setup to execute every time a transmit (TI) interrupt is requested.

The program would setup the serial port according to the application. The PTS vector at location 2056H would point to a QUAD word in the register RAM (IE: 01F8H) containing the PTSCB. The Source pointer in the PTSCB would point to the beginning message byte plus one (the first byte is sent manually). The Destination pointer in the PTSCB would point to the SBUF-TX special function register (location 1FBAh).

The PTS-COUNT would equal the number of byte to be transmitted minus one (the one that is send manually to start the SIO going).

The PTS-CONTROL is set for "Single transfer mode", Byte. Source is incremented and updated, and the destination is NOT increment or updated. (i.e., PTS-CTRL = 10010110B).

\[
\begin{align*}
\text{PTS-COUNT} & = (30 - 1) = 29 \\
\text{PTS-CONTROL} & = 9AH \\
\text{PTS-SOURCE} & = \text{Buffer} + 1 \\
\text{PTS-DEST} & = \text{SBUF-TX (1FBAh)}
\end{align*}
\]

The main line program set up the PTS, Port 2, INT-MASK1 bit for the TI interrupt, and the PTS-SELECT bit for the TI interrupt is also set. Lastly the Interrupts and PTS Interrupts are enabled through the EI and EPTS instructions.

To get the ball rolling the first character is sent to the SIO (SFR register SBUF-TX). When the first character is sent over the serial port, the TI interrupt requests a PTS cycle. The PTS cycle will read the next byte from the message buffer and write it to the SBUF-TX register. The Source pointer is incremented to point to the next byte of the message.

This continues until the PTS-COUNT is decremented to zero. When this happens, the PTS-SELECT bit is reset ("0"), and the PTS-SRV bit is set. This indicates to the core that all the PTS cycles have been serviced and need to be setup again. As soon as the PTS-SRV bit is set a normal software interrupt service routine for the TI interrupt is executed (depending on the interrupt priorities and pending interrupts).

This example is shown below.

### 2.2.3 BLOCK TRANSFER MODE

The block transfer mode is very similar to the single transfer mode. It too transfers data from memory to memory on interrupt requests. The difference is the number of transfers performed for each interrupt request.

The block transfer mode is able to transfer up to 32 bytes or words for each interrupt request. Since the PTS cycle cannot be interrupted, it is possible to have long latency times when using the block transfer mode. For example: if a block transfer mode is transferring 32 words from an external memory location to another external memory location, the latency could be as much as 500 to 600 states times.

Like the single transfer mode, the block transfer mode has a PTS-COUNT register (works identical to the single transfer mode).

PTS-SOURCE and PTS-DEST pointers that point to the source and destination addresses for the transfer. And a PTS-CTRL that identifies the PTS mode, word/byte, increment and/or update pointer bits.

An additional byte (PTS-BURST) contains the number of transfers to perform for each interrupt request. The maximum number of transfers per interrupt request is 32.
Program 1a. Send 30 bytes over the SIO using the PTS in Single Xfer Mode

; This program demonstrates the use of the serial port in
; single Xfer mode. The PTS is used to send data.
; This program sends 30 bytes of data out the serial port

 candidacy:

; windows for SRs

 CSEG AT 2068h ; sets up pointer to TI interrupt service
dw TI_ISR
CSEG AT 2065h ; pointer to PTS control block for TI
dw TI_PTS_CNTL

 Temp EQU 30H

RSEG AT 0E0H ; PTS transmit control block
TI_PTS_CNTL: DSB 1 ; transmit control register
TI_PTS_CNTL: DSW 1 ; transmit control register
TI_PTS_CNTL: DSB 1 ; source pointer
TI_PTS_CNTL: DSW 1 ; destination pointer

CSEG AT 2080H

; TI START:
; Set up port 2 and serial port control

 Ld R8, WSR, #1FH ; open window 1FH
 Ld R8, SP_CONCTRL, #01h ; serial port control
 Ld R8, SP_BAUD, #40071 ; baud rate = 9600 at 16MHz
 Ld R8, P2CON, #02h
 Ld R8, P2SEL, #08h ; select TxD and RxD peripheral
 Ld R8, P2IO, #00h ; RxD = input

 ; set up PTS transmit control block

 Ld R8, TI_PTS_CNTL, #0001010B ; load count
 Ld R8, TI_PTS_CNTL, #0011010B ; single byte transfer mode
 Ld R8, TI_PTS_CNTL, #01001010B ; beginning of the buffer
 Ld R8, TI_PTS_CNTL, #00010000B ; point to SBUF_TX

 Ld R8, INT_MASK1, #00000000B ; enable TI interrupts
 Ld R8, PTS_SELECT, #00010000B ; enable TI of PTS

 Ld R8, Temp [TI_PTS_CNTL] ; start transmission
 Ld R8, Temp SBUF_TX ; enable PTS

 ; enable interrupts

 EI

 ; The PTS will now handle the transmission of characters.
 ; When a PTS routine is done, the TI interrupt routine below
 ; will be called. This just notes that the PTS routine is finished.

 ; wait
Br Wait

 ; There is a bug with the interrupt server on the 196. It is possible for
 ; the wrong routine to be called. During the latency period for a normal
 ; interrupt, if a PTS interrupt occurs, the normal interrupt routine for the
 ; PTS interrupt will be erroneously executed.
Program 1b. Send 30 bytes over the SIO using the PTS in Single Xfer Mode

2.2.4 BLOCK TRANSFER MODE EXAMPLE

Suppose that each time an EXTINT rising edge interrupt happens, a set of 8 word registers is to be initialized with data from an EPROM table.

The code would initialize the EXTINT pin as special function/input, setup the PTS Control Block, and set the INT__MASK1, PTS__SELECT, EI, and EPTS bits to perform a PTS cycle for each EXTINT request.

The PTS Control Block would be as follows:

The PTS__BURST equals 8 (for the 8 word registers that need to be initialized each EXTINT request), The PTS__COUNT is set to the number of times before a software interrupt service routine would occur, in this case: 5 (there are 5 different tables in EPROM to be loaded into the same set of registers). The PTS__SOURCE contains the address of the first word EPROM location to be read, The PTS__DEST contains the address of the first word register to be initialized.

The PTS__CTRL byte contains "00001011". With this control, the block transfer mode is selected, it is set to transfer WORDs instead of bytes, the source pointer is both incremented and updated, while the destination pointer is only incremented not updated.

When the EXTINT request comes in, the PTS will transfer data from the EPROM table of 8 words to the 8 word register. Between each transfer, both the source and destination pointers are incremented.

When the PTS block transfer mode is complete, the PTS__COUNT is decremented and the new source address created by the PTS is placed in the PTS__SOURCE pointer (updated) and is pointing to: (EPROM address + 8). But, the destination address formed by the PTS is NOT placed in the PTS__DEST location. The PTS__DEST pointer remains pointing at the first word of the 8 registers.

When the next EXTINT request comes in, the next block of 8 words from the EPROM table is written to the 8 word registers.

The coded example is shown on the following page.
Program 2a. Using the EXTINT with the PTS Block Transfer Mode

```plaintext
$null
$include (macro.kr)
$lis:

; This program demonstrates the use of the EXTINT being used;
; with the PTS. Each time an EXTINT rising edge interrupts;
; the KR device, an 8 word block transfer PTS cycle initia-
; lizes and 8 word register bank.

; windows for SRFs

1f
P2SEL_W EQU 0C9h:byte ; window to 1fc9
P2REG_W EQU 0CDh:byte ; window to 1fcf
P2IO_W EQU 0C8h:byte ; window to 1fcf
CSEG AT 203Ch ; sets up pointer to EXTINT interrupt Service
    dcw EXTINT_ISR
CSEG AT 205Ch ; pointer to PTS control block for EXTINT
    dcw EXTINT_PTS_CNT
RSEG AT 0E0H ; PTS transmit control Block
EXTINT_PTS_CNT: DSB 1 ; Count register
EXTINT_PTS_Cnt: DSB 1 ; Control register
EXTINT_PTS_SRC: DSW 1 ; Source pointer
EXTINT_PTS_DST: DSW 1 ; Destination pointer
EXTINT_PTS_BST: DSB 1 ; Burst Count
DSEG AT 400H
RECS: DSW 8 ; 8 word registers that will get the data
CSEG AT 2080H
EXTINT_START: ; Set up part 2
    LdB WSR, #1FH ; Open window 1FH
    Ld SP, #0500H ; init stack
    orb P2SEL_W, #00000100B ; write out a 1 (open drain / input)
    orb P2REG_W, #04H ; Select EXTINT peripheral
    orb P2IO_W, #04H ; RxD = Input
    C1rB WSR ; set up PTS control block
    LdB EXTINT_PTS_CNT, #5 ; load count with 5 characters
    LdB EXTINT_PTS_CON, #0Bh ; Block/word/SU/SI/DI
    Ld EXTINT_PTS_SRC, #TABLE ; point to the beginning of the table
    Ld EXTINT_PTS_DST, #RECS ; point to 8 word registers
    LdB EXTINT_PTS_BST, #8 ; 8 word transfer per EXTINT request

; set up the interrupts
    LdB INT_MASK1, #01000000B ; Enable EXTINT interrupt
    Ld PTS_SELECT, #4000H ; Enable EXTINT of PTS
    EPTS ; enable PTS
    EI ; enable interrupts
```

20
; The PTS will now handle the transfer of information from 5 different
; EPROM tables, into 8 registers in the Code RAM. When the 5th one is loaded
; the PTS will reset the PTS_SELECT bit 14, and set the PTS_SRV bit 14, this
; will immediately execute a software EXTINT interrupt service routine.
; At this time the EPROM table pointer will be reset to TABLE and the Count
; will be reset to 5, and the PTS_SELECT bit 14 will be set again.
;
; wait:
;   Br     Wait
;
; There is a bug with the interrupt server on the 196. It is possible for
; the wrong routine to be called. During the latency period for a normal
; interrupt, if a PTS interrupt occurs, the normal interrupt routine for the
; PTS interrupt will be erroneously executed.
;
; Due to this bug, it is possible to enter these interrupt service
; routines before the PTS transfer is complete. Therefore it is necessary
; to check to see if the PTS_SELECT bit for a given routine is set. If
; it is, a PTS call should have been made, so the routine aborts and
; forces the call be setting the proper INT_PEND bit.

EXTINT_ISR:
   PUSH A
   JBS PTS_SELECT+1,6,ABORT_EXTINT
; Legal EXTINT interrupt service routine call - reset PTSCB.
   Ldb EXTINT_PTS_CNT, #5      ; load count with 5
   Ld EXTINT_PTS_SRC, #TABLE   ; point to the beginning of the table
   Or PTS_SELECT, #4000H        ; re-enable EXTINT to PTS
   POPA
   RET

; If here, BUG occurred
ABORT_EXTINT:
   ORB INT_PEND1,#40h          ; reset PEND bit
   POPA                      ; and return
   RET

Table:
   DCW 2222H,2222H,2222H,2222H,2222H,2222H,2222H,2222H
   DCW 4444H,4444H,4444H,4444H,4444H,4444H,4444H,4444H
   DCW 5555H,5555H,5555H,5555H,5555H,5555H,5555H,5555H

end
2.2.5 A/D SCAN MODE, PWM MODE AND PWM TOGGLE MODE

There are three other special modes of the PTS. Each are designed to work with specific peripherals on the 8XC196KR devices, but can be used with any interrupt source in the PTS_SELECT (except EPAINTx).

Each of these modes: A/D Scan Mode, PWM Toggle Mode and PWM Mode will be described later in this document. The A/D Scan Mode will be described in the A/D section and the PWM Toggle and PWM Modes in the EPA section.

2.3 PTS Latency Times

Since the PTS is simply an interrupt routine handled in microcode, it too has latency associated with its execution. As with normal software interrupts, the PTS has to wait until the current instruction has been processed before executing. The longest latency comes in within 4 states of the next instruction to be executed and the next instruction is a NORML (assuming that the PTS is enabled, and no non-interruptable block transfer PTS or BMOV instruction is to be executed).

That latency time is 43 state times.

The PTS cycle will be performed following the NORML instruction. Documented in Figure 2-5 is the amount of states required to perform the PTS cycle. This time INCLUDES the vector to the PTS Control Block.

2.4 Top 5 Issues with PTS

1. Make sure that the PTS control block is on QUAD word boundaries.
2. The PTS can not be used with the EPAINTx interrupt vector.
3. The PTS control block does not use windows (16 bit addresses only).
4. Setting up the PTS:
   1. Initialize PTS vector
   2. Initialize PTS Control Block
   3. Set PTS_SELECT bit
   4. Enable PTS

![Figure 2-5. PTS Interrupt Response Time](image-url)
3.0 UNDERSTANDING THE PORTS

The 87C196KR/KQ/JR/JQ devices all have bidirectional ports that double as special function peripheral ports (EPA, SIO, SSIO, A/D, etc.). When the device is reset, most of these ports (P2.7 is an exception) are configured as Low Speed I/O, Open Drain output, with a weak pull up. In order to use these ports as their special function, they MUST be configured.

All the ports (except Port 0) have the same internal design. Some of the signals that drive the port cell are different. Below is a discussion about configuring and using these ports as either LSIO or Special Function Peripheral.

3.1 Port 0

The analog input pins on the 8XC196KR device are also called the Port 0 pins. These pins are input only. The input structure is shown in Figure 3-1.

These port pins are input only and do not need to be configured if using them as analog input pins or digital inputs.

The input pins are sampled on Phase 1 and read into the bus on Phase 2. They have internal voltage clamping devices (ESD) as well as a 150 Ohm series poly resistor. (See Figure 3-1).

Because of the way the multiplexer is designed (for minimum A/D errors), the maximum input current on any analog input is 1 mA. With this spec the A/D will yield an additional error on adjacent channels. Example: Force 1mA on channel 4; Channels 3 and 5 will have additional LSB errors.

Digital inputs are read through the P0PIN register (BYTE location 1FDAH). The A/D section of this document discusses the special function A/D peripheral.

This port should be straightforward, as it exists on all prior MCS®-96 devices.
3.2 Port 1 / 2 / 6

These port pins are much different from the “quasi-bidirectional” ports seen on prior MCS*-96 devices. These are NOT “quasi-bidirectional” port pins. They do however, have some traits of quasi-bidirectional. (See Figure 3-2).

These pins have Schmitt trigger CHMOS inputs (with about 100mV of hysteresis and a $V_{IL} = 0.3V_{CC}$ and $V_{IH} = 0.7V_{CC}$) and CHMOS Outputs configurable as Open Drain or Push/Pull.

After the FALLING edge of RESET the signal PPU is active. The duration of the PPU is controlled by an R-C network that varies in width for different temperatures and process files, but it is at least 100 nS long. The transistor associated with the PPU signal is about 5-24K Ohms ($\sim 1mA / 5V$ drop). This pullup is used to charge pin loads before coming out of reset.

The active low RESET signal, will activate the WKPU signal. This signal stays active till the user program writes to the $P_xSSEL$ register associated with each pin ("x" stands for 1, 2, or 6).

The $\sim 150\Omega$ pullup that was present on the quasi-bidirectional ports is also present on the KR ports, but it can ONLY be turned off after RESET by writing to the $P_xSSEL$ register. The write to the $P_xSSEL$ register does the actual turning off of the WKPU signal, the data written makes no difference. Subsequent writes to the $P_xSSEL$ have no affect on the WKPU signal. On the next RESET low signal the WKPU signal will be turned on again.

This also means that the 1 to 0 input switching currents are at their worst case condition ($50\mu A$ max) when the WKPU signal is active. If the pullup is turned off, there basically is NO/Little switching current.
The combinational logic is used to define the output. After writing to the PxREG register and the PxIO register, the port is configured as either PUSH/PULL or OPEN DRAIN (Provided the internal weak pullup was turned off by writing to the PxSSEL register). Figure 3-3 is a truth table for the combinational logic.

The PxSFIO register is an internal register that the special function peripheral writes to control the port. It is not visible to the core or the user. Its contents are seen at the port pin if the port is configured for special function (PxSSEL = "1") and output (Push/Pull or Open Drain).

After RESET the PxIO register is "FFH" (Open Drain, Input). The PxREG is set to a "FFH" and the PxSSEL = LSIO. Even though the user accepts this as the port conditions, it is recommended that he at least configure the PxSSEL register after RESET in order to turn off the WKPU signal.

For example: If PORT 1 is configured as Low Speed Input port; write to the P1REG register ("FFH"), then write the P1IO register ("FFH" for open drain, Input), then finally write to the P1SSEL register ("00H" to select LSIO, and turn off internal WKPU signal).

Figure 3-4 is a list of all the special function registers associated with ports 0, 1, 2, 3, 4, and 6 with their respective absolute locations. These registers can be “windowed” via the WSR register in the core.
3.3 Port 3 / 4

On past MCS®-96 devices, these ports have been open-drain. This is still true for the KR device ports 3 and 4. They are the Address and Data bus as well as open drain input/output port pins.

The structure of this port is similar in layout to the structure of the other ports 1, 2, and 6. The differences lie in the inputs to the port circuitry (OE, PPU and WKPU). The ports (3 / 4) have a PxREG and a PxPIN SFR register (see Figure 4 for absolute locations), but it does not have a PxSSEL or a PxIO SFR register. This means that the ports CANNOT be structured as PUSH/PULL or open drain. The circuit is hard wired to configure these pins as OPEN DRAIN only.

As stated in the Port 1/2 and 6 section, the WKPU signal gets turned off by the write to the PxSSEL register. These ports don't have a PxSSEL register, therefore the WKPU signal is connected directly to the RESET signal. This means that while the RESET is active (low) the WKPU is also active. The WKPU signal is turned off when the RESET# goes inactive.

The PPU signal that is usually connected to an R-C circuit and triggered by RESET FALLING edge is NOT connected. The PPU signal is ALWAYS inactive.

The last thing to note about PORT 3 and 4 hardware is the OE signal. Since there is no PxIO bit associated with each pin of these ports, the OE signal is always an open drain configuration. (On ports 1/2/5/6 the OE is: PxREG AND PxIO, on ports 3/4 the OE is: PxREG AND PxIO with PxIO hardwired to a ‘‘1’’ only - OPEN DRAIN.

Port 3 has an alternative function of Slave Port. Later revisions of this document will include discussions of this function.

It is also possible that future KX core devices will have push/pull capability on ports 3 and 4, when used as ports (not as system bus pins).

3.4 Port 5

The structure for PORT 5 is very similar to that of Ports 1 / 2 and 6. It too has Schmitt trigger CHMOS inputs and CHMOS Outputs (See previous sub-sections). However, some of the input circuitry has been optimized for high speed input capabilities.

When the port is configured as a system function, the P5IO register has NO affect on configuring the port.

For example, if the P5.0 pin (ALE/ADV) is configured as a system function (ALE/ADV), the port will be configured as Push/Pull regardless of the value in the P5IO register. This is true for all of port 5 pins. The system function defines whether the port is push/pull output or an input pin. (Open drain output is not a feature of the system functions).

The RD (P5.3) is also special in that the QD and QU transistors are stronger for higher loading capabilities. WKPU and PPU signals are also present on port 5 and need to be dealt with accordingly by writing to P5SSEL) register to turn off the 150K Ohm reset pull-up.

After RESET the port 5 pins are configured as LSIO. If the EA is strapped to run from external memory, the ALE/ADV (P5.0) and the RD (P5.3) are configured as system functions.

Configuring these pins as system functions will allow the device to read the CCB0 (2018H) and CCB1 (201AH) from the external memory device.

The rest of port 5 will either be configured by the software program or by the CCB0/CCB1 reads.

For example: The READY (P5.6) pin comes out of reset as a LSIO not as READY. If the CCB0 and CCB1 data directs the 8XC196KR device to use the READY pin to control wait states and the IRC field is “111” (max wait states are defined by ready), the device could lock up if the READY pin were to continue being an LSIO pin.

In this case the internal logic would re-configure the READY pin as a system (READY) function. Hence, no lock up condition.

If other pins on Port 5 are used as system functions, the PxSSEL register must be written. This will turn off the WKPU signal and configure the port as either LSIO or system function.

Note that the SLPINT (P5.4) has a third function. If this pin is driven to a logical low on the rising edge of RESET, the KR device will enter the ONCE (ON-Circuit Emulation) test mode. This mode tri-states all device pins except power pins and XTAL1 / XTAL2.

For the above reason, it is recommended that the P5.4 (SLPINT) pin be used as an OUTPUT ONLY pin. The internal weak pullup will insure that this pin is high prior to RESET rising edge is not loaded.
3.5 Top 5 Issues With the Ports
(1) Setup the ports to match the applications needs:
   1. Write to PxREG
   2. Write to PxIO
   3. Write to PxSSEL
(2) P2.2/EXTINT, The external interrupt is a special port function. Therefore, EXTINT function is selected by setting PxSSEL bit 2.
(3) If the EA is strapped to run from external memory (low), the ALE and RD pins are hardwired to system functions.
(4) The 150K Ohms weak pullup resister is turned on by the RESET low signal, and turned off by the first write to the PxSSEL register.
(5) Beware of the SLPINT/P5.4 being driven low on the rising edge of RESET (this will enter ONCE mode).

4.0 SERIAL I/O PORT (SIO PORT)
The serial port on the 8XC196KR has one synchronous mode and three asynchronous modes. Both the receiver and the transmitter are double buffered. This allows for the reception of a second byte before the first byte has been read, and uninterrupted transmissions, respectively. The synchronous mode (MODE 0) is often used for shift register based I/O expansion. Mode 1 is an 8 bit asynchronous mode used for normal communications like RS-232C, while modes 2 and 3 are 9 bit data asynchronous modes which are specially designed for multi-processor communications.

The serial port on the 8XC196KR is capable of sending two distinct interrupts to the core; a receive (RI) and a transmit (TI) interrupt. There are separate mask bits for these two sources in the INT__MASK1 register which is located at 13H. These mask bits can be used to disable ("0") and enable ("1") interrupts to the core, but the RI and TI bits of the SP__STATUS register will be set regardless of the setting of the mask bits. There are separate RI and TI vectors which are located at 2038h and 2036h respectively.

4.1 Serial Port SFRs
Control and status of the serial port is accomplished by using five dedicated registers: the Serial Port Control register (SP__CONTROL) at location 1FBBh, the Serial Port Status register (SP__STATUS) at 1FB9h, the Serial Port Baud Rate Register (SP__BAUD) at location 1FBCb, the Serial Port Transmit Buffer Register (SBUF__TX) at location 1FBAh, and the Serial Port Receive Buffer Register (SBUF__RX) at location 1FB8h. These are all byte addressable registers except SP__BAUD. It is word addressable. A map of these registers is shown below in Figures 4-1, 4-2, and 4-3.
4.1.1 SP__CONTROL

The SP__CONTROL register controls various aspects of the serial port's operation. The lower two bits (bit 1 and bit 0) selected which mode the serial port is in: Mode 0 = 00, Mode 1 = 01, Mode 2 = 10 and Mode 3 = 11.

Bit 2 of SP__CONTROL controls whether parity is being used. When it is high (1), even parity for mode 1 or mode 3 is selected. NOTE: Parity cannot be enabled for mode 2.

Bit 3 is used to enable the receiver (RXD to SBUF__RX).

Bit 4 is used to determine the setting of the ninth data bit in modes 2 and 3 during transmissions. It is cleared after each transmission and must be reset for each data byte whose 9th bit is to be set.

Bits 7, 6, and 5 of the SP__CONTROL register is reserved and should be written as zeros for compatibility with future products.

4.1.2 SP__STATUS

The SP__STATUS register contains status information about the serial port. It is very important to keep in mind that when this register is read, the RPE, TI, RI, OE, and FE bits will be cleared. This mandates the use of a shadow register (see example program 3) if more than one bit is to be tested.

SP__STATUS 1FB9H:byte

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>RB8/RPE: Set if 9th bit set (no parity)</td>
</tr>
<tr>
<td>6</td>
<td>RPE: Set if parity enabled and parity error</td>
</tr>
<tr>
<td>5</td>
<td>RI: Set after last data bit received</td>
</tr>
<tr>
<td>4</td>
<td>TI: Set at End of last data bit sent</td>
</tr>
<tr>
<td>3</td>
<td>FE: Set if no STOP bit found</td>
</tr>
<tr>
<td>2</td>
<td>TXE: Set when byte is in SBUF__TX</td>
</tr>
<tr>
<td>1</td>
<td>OE: Set if overrun error occurred</td>
</tr>
</tbody>
</table>

Figure 4-2. SP__STATUS Register

Bit 2 is the Overflow Error bit, and is set when a new byte is loaded into SBUF__RX (by the receiver) before the previous byte has been read. This alerts the user that some information was lost because it wasn’t read in time. In this case, SBUF__RX will contain the last byte received. The old data is lost.

The Transmitter Empty (TXE) bit (bit 3) is set if the transmit buffer is empty and SBUF__TX register is able to take up to two bytes. TXE will be cleared as soon as the first byte is written to SBUF__TX register. Only one byte may be written to SBUF__TX if TI alone is set. The TI bit (bit 5) is set as soon as transmission of the last data bit is complete and so indicates that the transmitter is ready to take another byte.

Bit 4 is the framing error bit. It gets set when a valid stop bit cannot be found for a received byte.

Note: The 8XC196KR A-step device will not generate a framing error if the last data bit sent is a 1. The part needs to see a low to high transition in order to detect the stop bit. This is fixed on later steppings.

The RI bit (bit 6) is set after the last data bit is sampled. This happens approximately in the middle of the bit time. It should be noted that if the SIO port is run in loop-back mode (with the transmitter and receiver tied together), the transmit flag (TI) will be set approximately 1 bit time before the receive interrupt (RI) flag is set.

The Receive Bit 8 (RB8, bit 7) is used when the port is configured in modes 2 or 3. This bit is set when the 8th data bit is set (counting from 0). The other function of this bit is the receive parity error bit (RPE). This will be set if an even parity error was detected by the receiver.

Bits 1 and 0 of the SP__STATUS register are reserved. Any data in these bits is to be ignored.

4.2 Baud Rate Generation

The SIO has a dedicated Baud Rate Generator clock. Baud rates for all modes are determined by the contents of a word register (SP__BAUD) at 1FBCh. While this is a 16-bit register, only the lower 15 bits actually determine the baud rate. The MSB selects one of two sources for the input frequency to the baud rate generator. When it is set to a 1, the frequency on the XTAL1 pin is selected as a source to the baud rate clock. If it is 0, the frequency on the T1CLK pin (P6.2) is used. SP__BAUD is a write only register; it will read all 1’s. It can be updated via word writes to locations 1FBCh.
The value to be placed in SP_BAUD for a given baud rate depends on both the mode and clock selected and can be calculated as follows:

\[270873 - 51\]

Figure 4-3. SP_Baud Register Equations

The equations used are different for both the synchronous and asynchronous modes, and for the internal (XTAL1) or external (T1CLK) clocks. The maximum frequency on the T1CLK pin is 4MHz. The T1CLK input is NOT prescaled.

The following table lists various baud rates, the value to be programmed into SP_BAUD, and the error associated with the resulting baud rate.

<table>
<thead>
<tr>
<th>Baud Rate</th>
<th>SP_BAUD Value</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>9600</td>
<td>8340H, 8067H</td>
<td>0.16%</td>
</tr>
<tr>
<td>4800</td>
<td>8682H, 80CFH</td>
<td>0.16%</td>
</tr>
<tr>
<td>2400</td>
<td>8D04H, 81A0H</td>
<td>0.16%</td>
</tr>
<tr>
<td>1200</td>
<td>9A0AH, 8340H</td>
<td>0.04%</td>
</tr>
<tr>
<td>300</td>
<td>E82BH, 8D04H</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

Figure 4-4. Common Baud Rate Values

4.3 SIO Port Configuration

Before the SIO unit can be used, two port 2 I/O pins (P2.0 and P2.1) MUST be configured. This is handled by writing to the P2SSEL register (1FC9h), the P2IO register (1FCBh), and the P2REG (1FCDh). Note that these are byte registers (See port section for details).

Setting the P2SSEL.0 bit to a 1 tells the port logic that pin P2.0 is to be controlled using an internal special function source and not act as a general purpose I/O port pin. Clearing P2IO.0 causes pin 0 to become a push/pull output. Similarly, if the RXD pin is to be used as an input/output, bit 1 of P2SSEL, P2IO, and P2REG must all be set to 1. Writing a 1 to P2IO.1 configures pin P2.1 to become an input/open drain output. Forcing a 1 in P2REG.1 is needed to insure that the pull down associated with that pin is turned off. Thus, RXD may be used, in mode 0, as both an input and an output with an external pullup. The following code segment demonstrates how to set up port 2 for use with SIO.

```
/* PORT 2 SFR's used with windows
P2IO  W EQU 0E8H:BYTE ; Window to 1FCBh
P2Sel_W EQU 0E9H:BYTE ; Window to 1FCBh
P2REG_W EQU 0EDH:BYTE ; Window to 1FCDh

; Port Configure
LD B WSR, #7Eh ; Window 1f0-1fd to e6-ff
Or B P2Reg, #12h ; Force 1 at P21
Or B P2IO, #02h ; set P2.1 as IN/OD output
And B P2IO, #FFeh ; set P2.0 and P/P output
Or B P2Sel, #03h ; Set P2.0 and P2.1 for SIO
*/
```
4.4 Mode 0: Synchronous Communications

Mode 0 is a synchronous mode which is commonly used for shift register based I/O expansion. In this mode the TXD pin outputs the clock (a set of 8 pulses) and the RXD pin either transmits or receives the data. Data is transmitted/received 8 bits at a time with the LSB first.

Mode 0 can be entered by first setting up port 2 as described above, and then setting up the SIO by writing the desired baud rate to the SP_BAUD register and writing the proper control value to the SP_CONTROL register. P2.1 must have an external pullup attached as it is configured as an input/open-drain output.

Reception starts when a 1 is written to the REN (Receiver Enable) bit in the serial port control (SP_CONTROL) register. If REN is already high, clearing the RI flag will start a reception. After the reception is complete, the RI flag will be set and an RI interrupt will be generated if enabled. In order to avoid a partial unwanted reception, the receiver must be disabled by clearing the REN bit in the SP_CONTROL register before the RI bit is cleared. If the SP_STATUS register is read before writing to SP_CONTROL, RI will automatically be cleared, thus starting a new reception.

Starting a transmission in mode 0 requires writing a byte to SBUF_RX register. A set of 8 pulses will be sent out from the TXD pin, and the data will be sent out of the RXD pin. NOTE: This is the only mode for which RXD can be used as an output. After the data has been shifted out, TI will be set and a TI interrupt will be generated if enabled.

4.5 Mode 1: Standard Asynchronous Serial I/O

Mode 1 is a standard asynchronous communications mode. The data frame used is shown in Figure 4-5. It consists of 10 bits; a start bit, 8 data bits, and a stop bit. If parity is enabled (PEN = 1) then a parity bit is sent in place of the last data bit. Only even parity is supported on the 8XC196KR. Parity is also checked upon reception, and if an error is detected, RPE (Receiver Parity Error) in SP_STATUS is set.

The transmit and receive functions are controlled by separate clocks, but both clocks operate at the same frequency. The transmit clock starts as soon as the baud rate generator is initialized, but the receive shift clock is reset when a 1 to 0 transition is received (signaling a start bit reception).
4.5.1 SETTING UP MODE 1 OPERATION:

The following code demonstrates how to set up mode 1 with no parity at 9600 baud, assuming a 16MHz. Also, polling the RI and TI status bits with the use of a shadow register is demonstrated. It is very important to set up port 2 properly since the serial port shares pins with this port.

The next section of code sets up the serial port. First, the baud rate register, SP_BAUD, is programmed. The high bit is set, indicating that the internal clock, XTAL1, is to be used. The control register, SP_CONTROL, is set for mode 1, no parity, and the receiver is enabled.

Reading the SP_STATUS register will automatically clear various bits, so a shadow register is used because the contents are to be checked twice (once for RI and once for TI bits) so that they can be acted upon later. The above program illustrates several points about using a shadow register for the SP_STATUS register. First, the shadow register needs to be updated to reflect the current status of the SP_STATUS register. This is handled by ORing the shadow_status register with the SP_STATUS register. Second, it is important to clear the various flags in the shadow register after they have been acted upon. This is handled by clearing the RI bit in the shadow register after reading the data in the buffer, and by clearing the TI bit in the shadow register after transmitting new data.

Program 3. SIO Communication via Polling the SP_STATUS Bits (TI and RI)
4.5.2 SIO AND THE PTS

The final example in this section demonstrates the use of interrupts and the PTS with the serial port.

In this example both the RI and TI interrupts are being used with the PTS. Each PTS control block is programmed for the single transfer mode.

The RXD and TXD pins are strapped in loop back mode and the message being sent out the TXD is being received in the RXD and placed into external RAM at location 221Dh through 231Ch.

The serial port is configured for 9600 baud using an external XTAL of 16 MHz. The Port 2 pins are configured accordingly (TXD = push/pull, RXD = open drain-input, and both are special function selected, P2SEL = 3h).
Program 4b. Using the PTS with both the TI and RI Interrupts

```
00F0  332  RSEG  AT  0Fh  ;PTS receive control block
00F0  333  RL_PTS_CNT; DSB 1  ;receive interrupt count register
00F1  334  RL_PTS_CON; DSB 1  ;control register
00F2  335  RL_PTS_SRC; DSW 1  ;Source pointer
00F4  336  RL_PTS_DST; DSW 1  ;Destination pointer
338
2080  339  CSEG  AT  2080H
2080  340  TL_START:
2081  341  ;
2081  342  ; Set up port 2 and serial port control
2083  343  2080  B1FF14  344  LdW  WSR, #1FH  ;Open window 1FH
2083  345  A1004018  346  Ld  SP, #0500h  ;Init stack CODE RAM
2087  347  B10DBB  348  Id  SP_CONTROL_W, #0DH  ;mode 1 enable recv and parity
2088  349  A16780BC  350  Id  SP_BAUD_W, #6067H  ;baud rate = 9600 at 16MHz
208E  351  9102CD  352  orb  TREG_W, #02h  ;RxD = Input
2091  349  9102CB  350  orb  P210_W, #02H  ;RxD = Output
2094  351  71FEBC  352  orb  P210_W, #0FEh  ;TxD = Output
2097  353  910DC9  354  orb  T52SEL_W, #03H  ;Select TxD and RxD peripheral
209A  355  1114  356  Ld  3CB, WSR  ;
357
209C  358  C1EEF0  359  Ld  TL_PTS_CNT, #254  ;load count with 254 characters
209F  35A  B1FEA1  35B  Ld  TL_PTS_CON, #1001101B  ;Single byte transfer mode
20A2  35C  A10321E2  35D  Ld  TL_PTS_SRC, #TABLE  ;point to the beginning of the table
20A6  35E  A1B1FE4  35F  Ld  TL_PTS_DST, #SBUF_TX  ;point to SBUF_TX as destination
20AA  360  
20A8  361  set up PTS receive control block
20A8  362  
20A9  363  B1FFFD  364  Ld  RI_PTS_CNT, #255  ;Since entire recep done by PTS
20AD  365  B195F1  366  Ld  RL_PTS_CON, #1001101B  ;
20BA  367  A11D2F4  368  Ld  RL_PTS_DST, #BUFFER  ;point to destination buffer
20B4  369  A1B1F2F  36A  Ld  RI_PTS_SRC, #SBUF_RX  ;point to SBUF_RX as source
20BB  36B  811813  36C  Ld  INT_MASK1, #0000100B  ;Enable RI and TI interrupts
20BF  36D  A1003004  36E  Ld  PTS_SELECT, #1800H  ;Enable RI and TI of PTS
20C0  36F  B1FFD0  370  Ldb  0D0h, #0FPx  ;Flag for PTS routines complete
20C2  371  B1E31D3  372  Ldb  0D1h, [TI_PTS_SRC]+  ;Clears the buffer
20C3  373  C701B1FD1  374  Sb  0D1h, SBUF_TX  ;Start transmission
20C7  375  ETS  376  Enable PTS  
20CB  377  FB  378  EI  ;Enable interrupt
20CC  379  ;The PTS will now handle the transmission and reception of characters.
20CD  380  ;When a PTS routine is done, the TI (or RI) interrupt routine below
20CE  381  ;will be called. These routines just clear a flag to note that the
20CF  382  ;PTS routine is finished.
20D0  383  900DD0  384  cmpb  0D0h, #0Dh  ;wait around for PTS to finish
20D0  385  D7FB  386  Jne  ;Useful stuff done here
20D2  387  B1D14  388  Ldb  WSR, #7Dh  ;Activate window 7Dh
20D5  389  B101BB  38A  Ldb  SP_CONTROL_W, #010h  ;Enable receiver
20DB  38B  90BP7F  38C  Orb  07Fh, SP_STATUS_W  ;Read status of port
20DC  38D  B1FF0F  38E  Js  07Fh, P_error  ;Check for parity error
20DE  38F  3C7F6F  390  Js  07Fh, 4_Fr_error  ;Check for framing errors
20E1  391  1114  392  Ld  Cbr  WSR  ;
20E3  393  27FE  394  Br  $  ;
20E5  395  P_error  ;
20E5  396  27FE  397  Br  $  ;
20E7  398  Fr_error  ;
20E7  399  27FE  399  Br  $  
```

270873–23
Program 4c. Using the PTS with the TI and RI Interrupts
The PTS control blocks (both RI and TI) are initialized and the interrupt mask and PTS_SELECT bits are also set.

Lastly the program sends the first byte from the buffer to the SIO SBUF Transmit register. This starts the transmission/reception process rolling.

The TI PTS cycle sends the data out the SBUF_TX. The RI PTS cycle receives the byte and transfers it to external RAM locations 221D to 231Ch.

After the PTS transfers are completed a normal software interrupt request (for both RI and TI) is executed. This will flag to the main line program that the PTS is completed.

Lastly the receiver (RXD) is disabled, and the Parity and Framing errors are checked.

4.6 Modes 2 and 3: 9 Bit Communications Modes

Modes 2 and 3 are asynchronous 9 bit communications modes. In mode 2, parity can NOT be enabled. However, the 9th bit is used to determine whether or not a receive interrupt will occur. If the 9th bit being received is set, RI will be set and a receive interrupt will occur. This allows for a selectable reception link.

For transmission in mode 2, the state of the 9th bit is determined by the setting of bit 4 (TB8). If TB8 is 1, the 9th bit will be set. TB8 is cleared after each transmission and so it must be written before each ("1") transmission.

Mode 3 is similar to mode 2 except that parity can be enabled and that the receiver will generate an interrupt (set RI) every time a byte is received (independent of the state of the 9th bit).

Modes 2 and 3 work very well together in a multi-processor environment. The master processor will operate in mode 3. While the slaves will usually operate in mode 2. When the master wants to talk to a slave, it will first set the 9th bit high with the address byte. The slaves (operating in mode 2) will be interrupted and the one that is being addressed can switch to mode 3. The two processors can then talk (with the 9th bit clear), and the other slave processors will not be interrupted.

Setting up modes 2 and 3 is just like setting up the other modes. Port 2 must be set up, the baud rate written, and the control register programmed.

Below are two program segments (one for the master and one for a slave) which demonstrate the use of these modes.

NOTE: THE SEGMENTS ILLUSTRATED ARE JUST THAT. THEY ASSUME THAT THE PORTS ARE SET UP AS NEEDED.

```
; MASTER
CSEG
| Initial Start:          | ; Disable Interrupts
| DI                   | ;
| DITPS                | ;
| LD                   | SP=0000h ; Initialize Stack
| LDB                  | We=7Fh ; 1CO-1hf = e0hf
| Clr                  | Tinel

SIO Configure:
LDI Wr,07Dh ; 1a0-1b0f = e0hf
LD sp_baud_wr,0467h ; Enable 9600 baud
LD sp_control_wr,01Bh ; Mode 3, Rec En, TB8
CAL
LDB SBF, TX_W, #03h ; See if slave #3 is there

clrl
DINZ Timel,sm1
DJNZ Timel+1,sm1 ; Wait about 0.7 second
JNC CmpB 0BH, #03h ; correct slave?
NE
error
LDB slave #06H ; slave respond flag
; do whatever is needed to complete transaction with
; slave

nones:
CLR SMR ; No windows open
Self
SIMP Self
error
SIMP error

; SLAVE
CSEG at 208h ; set up RI interrupt pointer
DCW RI SR

| Initial Start:          | ; Disable Interrupts
| DI                   | ;
| DITPS                | ;
| LDB                  | mode,#02h ; we start in mode 2
| LD                   | SP=0000h ; Initialize Stack
| LDB                  | Wr=47Eh ; Window 1CO-1hf = e0hf

SIO Configure:
LDB Wr,#07Dh ; 1a0-1b0f = e0hf
LD sp_baud_wr,0467h ; Enable 9600 baud
LD sp_control_wr,#0007h ; Mode 2, Rec Enabled
ORB INT_MASK1,#10h ; Enable Rec int
EI
SIMP self

; RI Interrupt Service Routine
RI_ISR:

PUSHA
INCBI
CMBP 9th, #03h ; count times in interrupt
JE cont
CMPB 0BH, #03h ; 0BH: read id
JE not, check ID
CLRB 0BH, #03h ; if not, check ID
JNC not
JMP not
not:
LDB 0BH, #00Bh ; yes switch to mode 3
LDI mode,#03h ; note mode switched
LDB mode,#03h ; echo slave id number
INCBI 8th

270873–25
```
There are a few points which should be emphasized about the above program segments. First, the master always operates in mode 3. Second, the slave operates in mode 2 while it is idle, but switches to mode 3 when the master is talking to it. Finally, the master only sets the 9th bit when it is sending out an address of the slave that it wishes to communicate with. This is done so that the other slaves will not be interrupted unless an address is being broadcast over the serial link.

4.7 Top 5 Issues with the SIO

(1) When using the T1CLK as the clock to the serial port, make sure that the pin (P6.2) is special selected (P6SSEL bit 2 = 1).
(2) When using the RI and TI flags in the SP_STATUS register, use a shadow register in RAM.
(3) Make sure that the SIO port pins are set up according to the application:
   1. Write to P2REG
   2. Write to P2IO
   3. Write to P2SSEL
(4) Parity is not possible with Mode 2.
(5) SBUF_TX and SBUF_RX registers are separate register (unlike past 8096/80C196 devices) and are BOTH double buffered.

5.0 SYNCHRONOUS SERIAL I/O AND PERIPHERAL TRANSACTION SERVER

The Synchronous Serial I/O (SSIO) unit of the 8XC196KR has two identical channels, each capable of transmit and/or receive functions, with a shared baud rate generator. The SSIO will provide simultaneous bi-directional communications between synchronous serial I/O devices (as between two KR processors, or other serial peripherals). Listed below are a few of the many modes possible with the SSIO:

- **Master Transceiver** (half duplex mode, single SSIO channel)
- **Slave Transceiver** (half duplex mode, single SSIO channel)
- **Dual Channel Masters with common clock** (full duplex, lock-step synchronous)
- **Dual Channel Slaves with common clock** (full duplex, lock-step synchronous)
- **Dual Channel with different clocks** (master transmits clock, slave receives clock)

The two channels share a single baud rate generator. It internally provides baud rates from 15.75K baud to 2 Meg at 16MHz. Both channels can be used with the Peripheral Transaction Server (PTS) using the Handshake mode.

Both SSIO channels have distinct interrupts to the core (XFR0 and XFR1). Each channel has a separate mask bit in the INT_MASK1 register located at 13H. The mask bits are used to enable ("1") or disable ("0") interrupts to the core. However, the INT_PEND1 XFR0 and XFR1 bits, will be set regardless of the setting of the mask bits.

SSIO channel 0 interrupt vector through location 2032H and channel 1 through location 2034H.
5.1 SSIO Port SFRs

Control of the Baud Rate Generator is accomplished by using the SSIO_BAUD Register at location 1FB4H. This register includes an enable bit (bit 7, 1 = enable) and seven bits to select a baud rate (bits 6 through 0) from 15.75 KHz to 2 MHz using a 16 MHz crystal or 11.82 KHz to 1.5 MHz with a 12 MHz crystal. After reset this register is cleared, resulting in the generator being disabled with a clock value of 2 MHz (16 MHz crystal). Each SSIO channel can be clocked from an external source, through the SC0 or SC1 pins (slave mode).

While writes to the SSIO_BAUD register will enable and set the baud rate value, reads of this register will return the current state of the baud rate clock with bit 7 returning the clock pin current state and bits 0 to 6 returning the current down count in the generator.

Control registers SSIO_STCR0 or SSIO_STCR1 are used to configure the two SSIO channels. The following figure defines the functions for each bit:

Bit 7 defines the mode of the channel. A “1” will program MASTER, while a “0” will program SLAVE. This refers to whether the SSIO clock pin will either transmit (master) or receive (slave) a clock.

At this point it is important to note that the SC0 and SC1 pin is fed back internally to clock the channel. Even when in MASTER mode the clock is fed back. Because of this, the SCx pin MUST be configured as Special Function pins (P6SSEL bits 4 and 6 = 1). It is not possible to shift the data out without the SCx pin selected as special function.

Bit 6 controls whether the channel is to receive (“0”) or transmit (“1”) data in/out of the data pin (SD0, SD1).

Bit 5 controls whether bit 6 is to be toggled to the opposite state (“1”) after transmission or reception is complete, or left alone (“0”). Setting bit 5 will change the channel from an output to an input or vice versa, after each byte reception or transmission.

Bit 4 controls whether the channel is to be used as handshake (“1”) or as a normal shift register (“0”). See handshake mode for details.

Bit 3 controls transmission or reception enabling. If this bit is set (“1”) it will either transmit the data when the SSIO_STBx register is loaded, or enable reception of data in the SSIO_STBx register.

Bit 2 controls whether bit 3 is to automatically re-enabled (set) when reception or transmission is complete. With this feature, the user need only write to the SSIO_STBx register and the SSIO peripheral will transmit immediately.
The last two bits of this register are status bits. Bit 1 refers to data integrity (overflow/underflow). This bit is initialized by the user program, and updated by the core with each byte received or transmitted.

Bit 0 refers to the status of the SSIO__STBx register. The user code **MUST** initialize this bit. If the channel is a transmit channel, this bit set (1) means that the SSIO__STBx register is empty and waiting for data. If the channel is a receive channel, this bit cleared (0), means that the SSIO__STBx register is empty.

**Note that on RESET this register is cleared, setting the channel into a receive, slave mode, and the status of the SSIO__STBx is empty. If modified to a transmit channel, make sure the user code sets (1) bit 0, indicating that the buffer (SSIO__STBx) is empty.**

5.2 Example 1

In this first example, the SSIO channel 0 sends data out and that data is received on the SSIO channel 1 (loop back mode). A Baud rate of 2 MHz is selected.

Both channels are jumpered together (clocks and data pins). It is important to first setup the port 6 pins prior to any SSIO transmissions or receptions.

Since Channel 0 is sending data in a master mode, SC0 (P6.4) and SD0 (P6.5) are set to be push/pull output pins. Conversely, Channel 1 is receiving both a clock and data and sets SC1 (P6.6) and SD1 (P6.7) as open drain input pins.

Program 5. SSIO, Send One Byte

In the above example, the port pins are configured, and the SSIO channel control registers are set to values that would either transmit or receive information in the SSIO__STBx registers.

Since only one byte is being received, only the STE bit in the control registers were set once. The write to the SSIO__STB0 register will send that byte out the SSIO channel 0 to channel 1’s SSIO__STB1 register (since the two channels are in loop back mode).

5.3 Using the PTS and Handshake Mode

The SSIO Handshake Mode requires the output of the clock to be defined as an open drain and used with external pull up resistors.

The main difference between handshake and normal clock mode is the clock edge that the data is clocked in/out on (See Figure 5-2). In the normal mode, the clock is low, and on the rising edge data is clocked in. The falling edge of the clock is used to change the data. In the handshake mode, this is inverted. The clock is high, and on the falling edge the data is clocked in. The rising edge is used to change the data.

If the PTS is used to transmit data, there needs to be some signal that tells the transmitter that the receiver has received the transmitted data. If there were no signal, the PTS would continuously send data out, regardless if the receiver acknowledges the information or not.
For this reason, the handshake mode was implemented. When the transmission of data is complete, the master clock is left to float (this is why the clock MUST BE setup as open drain output). The receiver, upon reception of the last bit, will pull its clock pin low and hold it low until the SSIO_STBx register is read by the receive processors program. This acknowledges reception of data and that the SSIO_STBx register is ready to receive information.

The master SSIO channel will sense the release of the clock line by the receiver, and start the next transmission.

The following example uses open drain on both data and clock lines (only clock is required to be open drain in order to function in the handshake mode). The two SSIO channels are still in loop back mode.

Note that in Example 1 the push-pull and open drain modes were used. After reviewing the code from both Examples 1 and 2 the only difference is the set up of the P6IO register and SSIO Control Registers 0 and 1.

5.4 SSIO and the PTS

In Example 3 the Peripheral Transaction Server (PTS) and the SSIO moves data from the SSIO receive register to a block of memory using the PTS. Since the PTS is used to transmit and receive data, the handshake mode is used. (SSIO0 and SSIO1 are in loop back mode).

For reasons previously described, it is a good programming practice to use the handshake mode when using the PTS with the SSIO peripheral.

```assembly
; KR SSIO Master Transmit/Slave Receive Handshake
PREG EQU 00H:BYTE ; window to 1638H
PPIO EQU 00H:BYTE ; window to 1638H
PBBEL EQU 00H:BYTE ; window to 1638H
SBKR 0 EQU 00H:BYTE ; window to 1638H
SBKR 1 EQU 00H:BYTE ; window to 1638H
Ssm 0 EQU 00H:BYTE ; window to 1638H
Ssm 1 EQU 00H:BYTE ; window to 1638H
RESULT EQU 12H:BYTE
WSR EQU 014H:BYTE
;
.cseg
AT 2000H
LDW WSR, #1FH ; set window
LDW PREG, #0FH ; turn off pullup
LDW PPIO, #0FH ; set as inputs
LDW PBBEL, #0FH ; special function
LDW BAUD, #0FH ; 2 Mbps, baud
LDW SSKR 0, #0DH ; master_transmit control
LDW SSKR 1, #18H ; slave_receive control
LDW SSIO_0, #53H ; data placed in xmit register

D_WAIT:
IN SSCR URLD_WAIT, wait for data received
STB SSIO_STB; RESULT
Smp 5

; Program 6. SSIO, Send Byte in Handshake Mode
sinlude (Macros)

Ssm 0 EQU 1F8H:BYTE
Ssm 1 EQU 1F8H:BYTE

RSEG AT 2000H
LDW P6IO, #0FH ; PTS control block
PTS Ssm 0_CNTL DBH 1 ; Count value
PTS Ssm 0_CNTL DS 1 ; Control for PTS
PTS Ssm 0_CNTL S 1 ; Source Pointer
PTS Ssm 0_CNTL D 1 ; Dest Pointer
S fkk_BYTE D 1 ; Source Data
PTS Ssm 1_CNTL DB 1 ; PTS control block
PTS Ssm 1_CNTL DS 1 ; Control value
PTS Ssm 1_CNTL S 1 ; Source Pointer
PTS Ssm 1_CNTL D 1 ; Dest Pointer
DBT_BYTE D 1 ; destination byte

CSEG AT 2000H
DCW PTS_Ssm 0_CNTL ; points to PTCB for Ssm 0
DCW PTS_Ssm 0_CNTL ; points to PTCB for Ssm 1

CSEG AT 2002H
DCW Ssm 0StateToProps ; software ISR for Ssm 0
DCW Ssm 1StateToProps ; software ISR for Ssm 1

; Main Program
;
: Send data from the source byte to the destination byte through the SSIO channels using the PTS.
;
: SSIO sends the information. SSIO 1 receives it.
: Handshake mode is used in order not to loose any transmitted information.

CSEG AT 2000H

Ssm 0 EQU 500H ; SP = Code RAM
LD SP, #500H
LD H WSR, #1FH ; Open window 1FH
```

270873–31

Program 6. SSIO, Send Byte in Handshake Mode
Program 7. SSIO and the PTS

In the above example several things should be noted. Data is sent out the SSIO0 and received in SSIO1 at 50K baud.

Channel 0 is setup to transmit the first byte, then wait for the receive channel to read the SSIO0_STB1 to transmit the next byte (this is accomplished through the SSIO Handshake mode and PTS interrupts on both XFR0 and XFR1).

The PTS is sending information from the SOURCE_BYTE to the DEST_BYTE at 50k baud.

After 255 transfers and receptions, a normal software interrupt is generated. This software interrupts just starts the process all over again.

The most recent data sent and received will be in SOURCE_BYTE and DEST_BYTE.

One way to delay the processor from sending out another byte would be to NOT use the PTS to receive the byte. The transmit could still be used with the PTS, but the next byte would not be sent until the first byte is received. (Also using the handshake mode).

Using the normal software interrupts to receive the information and the PTS to send the information means that the processor can still be left to do other processing with higher baud rates of communication.

5.5 Top 5 Issues with the SSIO

(1) Sending Data

1. Setup Port 6
2. Set SSIO_BAUD
3. Set Control with STE bit set
4. Send by writing to STB

5. Setup Port 0
6. Set SSIO_BAUD
7. Set Control with STE bit set
8. Send by writing to STB

In the above example several things should be noted. Data is sent out the SSIO0 and received in SSIO1 at 50K baud.

Channel 0 is setup to transmit the first byte, then wait for the receive channel to read the SSIO0_STB1 to transmit the next byte (this is accomplished through the SSIO Handshake mode and PTS interrupts on both XFR0 and XFR1).

The PTS is sending information from the SOURCE_BYTE to the DEST BYTE at 50k baud.

After 255 transfers and receptions, a normal software interrupt is generated. This software interrupts just starts the process all over again.

The most recent data sent and received will be in SOURCE_BYTE and DEST_BYTE.

One way to delay the processor from sending out another byte would be to NOT use the PTS to receive the byte. The transmit could still be used with the PTS, but the next byte would not be sent until the first byte is received. (Also using the handshake mode).

Using the normal software interrupts to receive the information and the PTS to send the information means that the processor can still be left to do other processing with higher baud rates of communication.

5.5 Top 5 Issues with the SSIO

(1) Sending Data

1. Setup Port 6
2. Set SSIO_BAUD
3. Set Control with STE bit set
4. Send by writing to STB
(2) Receiving Data
1. Setup Port 6
2. Set SSIO__BAUD
3. Set Control with STE bit set

(3) Using the PTS with high baud rates will require a
great deal CPU time.

(4) Initialize both the TBS and OUF bits accordingly
when writing to the SSIO__STCRx registers.

(5) When using the PTS, use the Handshake Mode
only.

6.0 ANALOG TO DIGITAL CONVERTER

The analog to digital converter on the 8XC196KR is
very versatile. It can perform an 8-bit or 10-bit conver-
sion, perform voltage threshold detection, and has two
test modes for converting on Analog GND and VREF.

There are 8 input channels which are multiplexed to
the converter. The sample and conversion times for any
channel are programmable, allowing the part to per-
form fast conversions at any frequency of operation.

The A/D has 4 dedicated SFRs which control its oper-
ation. These are the AD__COMMAND, AD__TIME,
AD__TEST, and AD__RESULT registers, located at
1FACH, 1FAFh, 1FAEh, and 1FAAh respectively.

6.1 A/D Command Register (AD__COMMAND)

Figure 6-1 shows a diagram of the AD__COMMAND
register. The lower three bits select the channel to be
converted. Bit 3 determines when the conversion
should start. If cleared, the conversion will be started
by the Event Processor Array (EPA), if set the conver-
sion will start within 3 state times of writing to the
AD__COMMAND register.

Bits 4 and 5 are the mode bits. Bit 4's function is depen-
dant upon whether the A/D is performing a normal
conversion, or is in threshold mode. If the A/D is per-
forming a regular conversion, bit 5 should be cleared
and bit 4 will determine 8- ("1") or 16-bit ("0") conver-
sions.

With bit 5 set, the threshold mode is selected. Clearing
bit 4 tells the A/D to detect when the value is above the
threshold value; setting bit 4 indicates a value below the
threshold should be detected.

The upper 2 bits of the AD__COMMAND are res-
erved and should be written as "00" to maintain com-
patibility with future products.

If the AD__COMMAND register is written while a
conversion is being performed, the old conversion will
be aborted and a new one will start. The A/D done bit
in the AD__RESULT register will NOT be set until the
completion of the second conversion.

6.2 A/D Time Register (AD__TIME)

The AD__TIME register, shown in Figure 6-2, allows
the sample window and conversion time (per bit) to be
optimized by the user depending on the clock speed of
the '196KR. The Sample time controls how long the
analog input voltage is connected to the sample capac-
itor. It must be long enough to properly charge the sam-
ple capacitor, but if it is too long, the input voltage may
change and introduce error. The Conversion time de-
fines the length of time to convert one bit. It needs to be
long enough to allow the comparator to settle, but can-
not be to long or the sample capacitor will discharge
and introduce error.

The upper 3 bits of this register program the sample
time (SAM), while the lower 5 bits program the conver-
sion time (CONV).
Figure 6-2. AD TIME Register

Sample time (in states) = \(4^*\text{SAM} + 1\)
Conversion time (in states) = 
\[
\# \text{ of bits} \times (\text{CONV} + 1) + 1.5
\]

In order to guarantee an A/D specification of ±3LSBs, the sample time should be at least 3.5\(\mu\)S and the conversion time should be at least 16.5\(\mu\)S. (Consult the latest specs for the most current values.)

The AD TIME register could be programmed to do a conversion in less than 4us, but the results would be about 200mV off.

Valid ranges for the SAMple window are 1-7 and the CONVersion timer should be between 2 and 31 inclusive. This yields a valid range for the AD TIME register of 22h-FFH. The AD TIME register should never be written with all zeros.

Assuming that the sample time is set such that the sample capacitor charges properly. The accuracy will still be a function of conversion time as show in Figure 6-3. This graph was obtained by testing several (typical) devices across automotive temperature range (−40° to +125°). Note that the conversion accuracy drops off VERY rapidly for conversion times under 10\(\mu\)S.

As an example consider an 8XC196KR running at 16Mhz. This gives a state time of 125ns, which means that to meet minimum specs on a 10-bit conversion, the SAM should be programmed with a 7h and CONV should be 0Dh (EDh should be placed in AD TIME.)

Note: When determining the sample time, it is extremely important for the user to consider the input circuitry associated with the channel being converted. The circuitry must be able to supply enough current to charge a 2 pF capacitance with a 1\(\mu\)A leakage current in the specified time.

6.3 A/D Test Register (AD TEST)

The AD TEST register, shown in Figure 6-4, can be used to enable two test modes, and to modify the zero offset of the A/D. Conversions can be performed on either Analog Ground or VREF, to gain insight as to the transfer function of the A/D. Small amounts of zero offset error can be corrected by using the offset adjustment.

For compatibility with future products, the command register should be written to select channel 7 when performing conversions on VREF or A GND.

Figure 6-3. A/D Error vs. Conversion Time

Figure 6-4. AD TEST Register
Bit 0 of AD__TEST is the test enable bit. It should be clear to perform normal conversions on any of the eight analog inputs, and set to perform conversions on $V_{REF}$ or $A_{GND}$.

Bit 1 selects between $V_{REF}$ ("1") or $A_{GND}$ ("0"). Bits 2 and 3 control the offset adjustment as shown in Figure 6-4. Later a sample program will be presented which demonstrates how to adjust the offset. Small amounts of offset errors, both negative and positive, can be adjusted through these bits. When the AD__RESULT register is read, the effected result (after offset adjustment) will be reflected, automatically.

The upper four bits are reserved and should be written as all 0's.

Figure 6-6 shows a typical low temperature (−40°C) graph of absolute error vs input voltage. This depicts many different types of errors inherent in the conversion process. The zero offset error can be seen, about 4mV, as well as the full-scale error, about 8mV.

Another effect which generally only shows up at low temperatures is a “bowing” S curve that can be seen across the entire transfer function. This is caused by small amounts of noise accumulated across the entire resister ladder.

At higher temperatures, bowing disappears, leaving only the stair-step or saw tooth effect. This is usually caused by errors in the resister ladder when the values “foldback” to conserve space.

6.4 A/D Result Register (AD__RESULT)

The last SFR is the AD__RESULT register shown in Figure 6.5. The lower three bits in this word register contain the channel number that was converted. Bit 3 is the BUSY bit and is set approximately 8 state times after a conversion is started. Bits 4 and 5 are reserved (ignore read value).
Figure 6-6. A Typical A/D Transfer Function Error, with Offset and Full Scale Errors
6.5 Example A/D Programs

In this section, a few examples will be presented on using the A/D. The first program segment shows how to start a 10 bit conversion on channel 5. The AD_TIME register is set-up, followed by writing to the AD_COMMAND register. The AD_RESULT register is then polled until the conversion is complete.

6.5.1 USING THE A/D WITH THE PTS

The Peripheral Transaction Server (PTS) can be used with the A/D, allowing up to 256 conversions without CPU intervention. In the example program below, the PTS is used to perform 8 10-bit conversions (one for each channel).

```
R0 EQU 00h:WORD .00 register
AD_Time_W EQU 0EFh:BYTE .Window to AD_Time register
AD_Test_W EQU 0EEh:BYTE .Window to AD_Test register
AD_Result_W EQU 0EAh:WORD .Window to AD_Result register
AD_Command_W EQU 0ECh:WORD .Window to AD_Command register

CSEG AT 2080H

TI_START:

LdB WSR,#7Dh ;window 1FA0-1FBF to 80-FF
LdB AD_Time_W,#0Ef ; 3.3 us samp/20us convert
LdB AD_Test_W,#06h

LdB AD_Command_W,#0Ch ;start conversion, ADCH5
Ld R0,R0
 ;dummy statements to wait for
Ld R0,R0
 ;BUSY bit to get set

wait:
Jb5 AD_Result_W,3,wait
St AD_Result_W,80h ;save it

Figure 6-7. Program Segment to Initialize A/D and Convert on ACH5

The A/D scan mode in the PTS is used to perform this function. It operates as follows:
1. The word pointed to by PTS_SD is read. The lower byte is saved in a temporary location within the ALU.
2. PTS_SD pointer is incremented by two.
3. The register pointed to by PTS_REG is read and stored at the location pointed to by PTS_SD.
4. PTS_REG is then incremented by two.
5. The value that was saved in (1) is stored in the register pointed to by PTS_REG.
6. PTS_SD is optionally updated.
7. PTS_COUNT is decremented.
When PTS\_COUNT reaches zero, the PTS\_SELECT bit for the A/D is cleared and the PTS\_SRV bit for the A/D is set causing a normal interrupt to happen.

The PTS\_REG is set up to point to the AD\_RESULT register (It can only point to Register RAM or SFR locations), thus it is read every PTS cycle, and when PTS\_REG is incremented by two, it points to the AD\_COMMAND register which is then written.

Since the last thing that the PTS cycle does is a load into the AD\_COMMAND register (to start another conversion), the last value in the A/D result RAM table should be loaded with 0000h. This will NOT start an A/D conversion when the last channel is complete (Remember that the process starts with a conversion started manually).

The table format is shown in Figure 6-8:

The routine starts off by initializing AD\_TEST and AD\_TIME. The PTS control block is then setup with PTS\_REG pointing to AD\_RESULT and PTS\_SD pointing to the start of the table. The next block of code sets up the table by filling all of the A/D command slots.

For simplicity in coding, each A/D channel is done in succession (7 ≥ 0). However, any order conversions can be performed, as well as any bit combination (10- or 8-bit conversions).

Interrupts for the A/D are masked (enabled) in both the core and PTS. Then the A/D is started by writing to the AD\_COMMAND register (starting an A/D on channel 7).

After the first sample is completed, the PTS reads the result and stores it in a table. It then loads the next command, from the table, into the AD\_COMMAND register. This is repeated until 8 values have been read. Note the that last command (for the 8th read/write) is a “dummy” command and does not start another A/D conversion, as the A/D doesn’t need to be restarted.

After 8 samples are collected, the A/D interrupt service routine is called and the data table is “cleaned up” by shifting all of the data words to the right by 6. This leaves just the sample value in the data table.
$nolist
$include(kr.inc)
$list

This code samples A/D channels 0-7 in succession using the PTS scan mode. The data is stored in table, as follows:

A/D_Command1 A/D_Data0 A/D_Command2 A/D_Data1 ...
A/D_Command7 A/D_Data6 0000 A/D_Data7

The '0000' in place of the last command for the A/D is needed so that another, unwanted, conversion will not be started.

AD_Time_W EQU 0E7h:BYTE ;Window to AD_Time register
AD_Test_W EQU 0E8h:BYTE ;Window to AD_Test register
AD_Result_W EQU 0E9h:WORD ;Window to AD_Result register
AD_Command_W EQU 0EAh:WORD ;Window to AD_Command register

TBL EQU 082h:WORD ;Pointer into command/data table
TEMP EQU 084h:WORD ;Temporary storage
TABLE EQU 3000h:WORD ;Start of command/data table

CSEG AT 204Ah ;Pointer to AD PTS command block
DCW PTS_COUNT

CSEG AT 200Ah ;Set up AD_ISR pointer
DCW AD_INT

RSEG AT 070h ;Allocate AD PTS command block

PTS_COUNT: DSB 1
PTS_CONTROL: DSB 1
PTS_SD: DSW 1
PTS_REG: DSW 1

CSEG AT 2080H
T1_START: DI

OPTS
Ld8 PTS_COUNT,#08h ;8 total conversions
Ld8 PTS_CONTROL,#0CBh ;A/D scan; SUSLDI
Ld PTS_SD,#Table ;Point to data table
Ld PTS_REG,#AD_Result ;Point to AD_result SPR

Program 8a. A/D Scan Mode using the PTS
Program 8b. A/D Scan Mode using the PTS

; Fill Command table

  Ld TBL,#TABLE
  Ld TEMP,#0000h  ; Command for starting conversion, channel 6
setups:  St TEMP,[TBL]+  ; fill command bytes in table
  Inc TBL  ; skip data word in table
  Inc TBL  ; (result word)
  Dec TEMP  ; decrement command by 1
  Cmp TEMP,#0007h
  Jne setup  ; stop when TEMP = 0007h

; NOTE: After the last channel is read by the PTS, the A/D should NOT
; be restarted. Therefore the last command entry in the table should
; be a dummy and not start a conversion!

  St R0,[TBL]  ; No new conversion
  OrI INT_MASK,#20h  ; enable AD done interrupt
  OrI PTS_SELECT,#20h  ; enable AD done to PTS
  LdI AD_Command_W,#07h  ; start conversion, ADCH7
  Eli
  EPTS
self:  Sjmp self

; A/D Interrupt Service Routine

AD_INT:
  Pusha
  Jbs PTS_SELECT,5,abort  ; check validity of enterance
  ; The following code "cleans up" the data from the AD.
  ; The lower six bits are discarded, leaving only the 10 data bits.

  Ld TBL,#TABLE+2  ; start of data
  LdI 80h,08h  ; count

fix:  Ld TEMP,[TBL]
  Shr TEMP,#06h  ; get rid of junk
  St TEMP,[TBL]+
  Inc TBL  ; skip command word
  Inc TBL
  Dj nz 80h,fix
  Popa
  Ret

; If we got here due to the bug in the interrupt handler
; force a call to the PTS routine, as this should have been called
; in the first place.

abort:  OrI INTPEND,#20h  ; force call to PTS
  Popa
  Ret
end
6.6 Threshold Detection

The threshold mode on the A/D allows the CPU to be notified when the value on one of the A/D channels crosses either above or below a predetermined value. The following code demonstrates one possible application of this A/D mode.

The A/D is set up to monitor a channel and generate an interrupt when the value on it passes above 2.5 V. The conversion is started and the 8XC196KR is put into idle mode. When the A/D determines that the value on ADCH0 is greater than 2.5 V, an interrupt is generated and the CPU exits the idle mode.

```
; The window equates are the same as before
CSEG AT 200AH
DCW AD_ISR
CSEG AT 2060H
TI START:
D
O:B INT MASK,#20h ; enable a/d interrupt
LdB WSR,#70h   ; FAD9-FF to E0-FF
Ld# AD Time_W,#00fH ; 3.3us samp/3us convert
Ld# AD Test_W,#00h ; normal conversion
; Write to RESULT (Ach > 2.5 volts)
; Upper 10-bits the RESULT sets the threshold value
Ld# AD Command W,#28h
Jmp idle ; enter idle, wake up when a/d done
; continue executing when A/D sees >2.5v on ADCH0
```

When the A/D conversion value crosses above the 2.5 volt level, set by the upper byte of the AD_COMMAND register. The device will exit idle mode and execute the AD_ISR interrupt service routine.

6.7 A/D Test Modes

The A/D on the 8XC196KR can perform conversions on AGND or V_REF. This allows for the software detection of offset and full-scale/gain errors. Small amounts of offset errors can be adjusted by writing to the AD_OFFSET register (bits 2/3).

The following code segment attempts to minimize the offset error. The method used is very straightforward. The routine is called by the main routine. It starts by assuming that a ~ 5.0mv offset is the best adjustment for offset correction. It then performs 16 conversions on AGND, summing the results. If the sum is more than 8 then the next higher offset value is tried. This is repeated until an offset with less than 8 on 16 conversions is found, or, if none can be found, + 2.5mv is used. The code will return to the main routine with the best offset value in the AD_OFFSET register.

```
; The window equates are the same as before
EQU DOR,WORD ; sum of conversion results
count EQU 07h,BYTE ; count for conversions
EQU 04h,WORD ; result of last conversion

CSEG
AD_OFFSET_ADJUST:
Ld# INT MASK,#00h ; disable A/D interrupts
Ld# WSR,#70h ; FAD9-FF to E0-FF
Ld# AD Command W,#00fH ; 23.3us convert
Ld# AD Test W,#00h ; Convert AGND 5.00mv check
Ld# AD Test W,#00h ; Convert (AGND-2.5mv)
Ld# AD Test W,#01h ; Convert (AGND+0.0mv)
Ld# AD Test W,#05h ; +2.5mv offset wins
Jmp exit
check:
Ld# AD Command W,#00h ; 10-bit conv. on ch?
Jt #0,080 ; dummy to wait for
Ld #0,080 ; BUSY bit to be set
wait:
Ld# AD result W,#3 ; wait till done
Jt #0,AD Result W
Shr #0,08h ; just want result
Add #0,0 ; sum results
Dnc #0,0 ; loop if not done
Ld #0,08h ; arbitrary cutoff
Jt #1,exit
; if sum is less than 8, then return to main routine by
; POPing the old Program Counter off the stack and
; returning
Tb #0,08h ; return to main
exit:
```

270873–37
6.8 Top 5 Issues with the A/D
(1) Small amounts of offset errors can be adjusted using the AD__TEST register.
(2) AD__COMMAND and AD__RESULT are two separate SFRs.
(3) Threshold Detect Mode of the A/D does a continuous conversion on one channel.
(4) Conversion times of less than 16uS and Sample times of less than 3.5uS will not produce less than 3 LSB error.
(5) In the AD__TEST register, the LSB bit is the ENABLE bit for the Convert on A GND or V REF. Bit 1 is the control bit for which to convert (1 for V REF 0 for A GND).

7.0 EVENT PROCESSOR ARRAY (EPA)

7.1 Timers
The EPA has two 16-bit timers/counters, which are controlled by two registers, TIMER1__CONTROL and TIMER2__CONTROL at 1F98h and 1F9Ch, respectively. The values of the timers can be read at 1F9Ah and 1F9Eh (TIMER1 and TIMER2). The maximum count rate of each timer is based on the internal clock rate divided by 4. This yields a resolution of 250ns at 16MHz. In addition, TIMER2 can be programmed to count when TIMER1 over/under flows, allowing a 32-bit counter to be formed.

Although the EPA can not capture or compare on 32-bit values, an EPA channel can be programmed to interrupt on any number of TIMER1 overflows. In addition, TIMER2, with the increment on overflow, can still be prescaled.

Both timers can be clocked by an internal or external clock with internal or external direction (Up/Down) control. Quadrature clocking is also available, allowing for easy interfacing with an encoder wheel.

The timers do not have auto-reload features, but encoder wheel interface can be achieved through a small amount of PTS interface. This will not be demonstrated in this paper.

Timer overflows are mapped into the EPAINTx interrupt in the core. However, each timer has its own mask and interrupt pending bit in the EPA__MASK1 and EPA__PEND1 registers.

Figure 7-1 shows a map of the TIMERn__CONTROL register. Bit 7 is the timer enable bit, and controls whether the timer is active. Bit 6 is the internal direction control (Up = 1, Down = 0).
The next three bits (bits 5–3) are the mode control bits. They determine if the clock and direction control should be internal or external, or if quadrature clocking should be used.

The three least significant bits control the prescale to be applied to the clock. These range from a divide by 1 (internal clock/4) to a divide by 64 (internal clock/256). This prescale is applied to both internal AND external clocks. NOTE: When using an external clock, the timer will count on EACH edge of the clock (assuming no prescaling is in effect).

The time registers (TIMER1 and TIMER2) are both readable and writable. This allows for more flexibility in the generation of interrupts.

### 7.1.1 TIMER EXAMPLES

The code segment below shows how to set up a software timer which will generate an interrupt in 1 ms. TIMER1 is first loaded with 1000 (decimal). Then it is programmed to count down, once per microsecond. The EPA__MASK1 is set up to allow interrupts on TIMER1 overflows or underflows.

The interrupt occurs when the TIMER1 rolls under 0000. The interrupt service routine for EPAINTx will have to determine which source caused the interrupt and take whatever action is needed. An example of this is given later in the EPA outputs section.

```assembly
    temp equ 070h:word
    Csec at 2500h

Ld temp, EPA__MASK1
Or temp, #010h
St temp, EPA__MASK1
OrB INT__MASK, #01h
St INT__MASK, #01h
Ld temp, #1000
St temp, TIMER1
Ld #0100
St temp, #0100
Ld #0000
St temp, TIMER1
OrB temp, #082h
St temp, TIMER1
OrB temp, #0100
St temp, TIMER1
OrB temp, TIMER1__CONTROL
Br $;wait for underflow
```

### 7.2 EPA Input/Output Structure

The EPA section has ten (10) Capture/Compare modules which each support timed event input and output for a single pin. There are also two Compare only modules (COMP0 and COMP1) which share their outputs with two of the EPA channels (EPA8 and EPA9 respectively). The Capture mode can be used to generate an interrupt on an input edge, reset the opposite time base timer, start an A/D conversion, or simply capture the time a transition of its input pin occurred.

The Compare function is for output time events. It can change the state of its output pin when its time base timer matches the value in its EPA__TIMEEn register. Also, an EPA channel has the option of resetting its own timer as well as the opposite timer, or start a timed A/D conversions.

There are two dedicated SFRs for each EPA channel that control the operation. These are EPA__CONTROLn, and EPA__TIMEEn registers. The EPA__CONTROLn is detailed in Figure 7-2.

The EPA__CONTROLn set of registers are used to configure their associated pin. The bit map of the control register is as follows:

```
+----+----+----+----+----+----+----+----+
| 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
+----+----+----+----+----+----+----+----+
| RM | TB | CE | M1 | M0 | RE | AD | ROT | ON/RT |
+----+----+----+----+----+----+----+----+
RM - “1” Enables Remapping (EPA1 and EPA3 Only)
TB - “0” = Timer1, “1” = Timer2
CE - “1” Enables Comparator
RE - “1” = Lock Time Entry
AD - “1” = Start A/D Conversion
ROT - “0” = Same Timer as TB, “1” = Opposite
ON/RT - Overrun/Reset Timer Enable
```

**Figure 7-2. EPA__CONTROL Register**
RM  Bit 8 of EPA(CONTROL). This only has an effect on channels 1 and 3. Setting this bit enables the lower adjacent channel to set/reset/toggle the channels output pin. This allows channels 0 and 1 to control the same output (channel 1’s), or channels 2 and 3 to control output on channel 3. Clearing this bit disables the remap feature.

*EPA(CONTROL1 and EPA(CONTROL3 must be written as WORDs.*

TB  Bit 7 is used to select which timer should be used as the time base for captures or compares. A “0” selects TIMER1, and a “1” selects TIMER2.

CE  Bit 6 selects between capture and compare modes: “0” selects capture mode while “1” enables the comparator. By enabling the capture function, only generates an interrupt when modes 1, 2, or 3 is used.

MX  Mode bits. Bits 5 and 4 are the mode select bits. They operate as follows:

<table>
<thead>
<tr>
<th>M1</th>
<th>M0</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>no operation</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>capture on + edges</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>capture on – edges</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>capture +/– edges</td>
</tr>
</tbody>
</table>

In Capture Mode

<table>
<thead>
<tr>
<th>M1</th>
<th>M0</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>no operation</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>reset output pin</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>set output pin</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>toggle output pin</td>
</tr>
</tbody>
</table>

RE  Bit 3 can be used to “lock” a Compare function. When set to “1”, the compare function is always enabled. When clear, the event will occur only once, and then the EPA(TIME register must be rewritten.

AD  Bit 2, when set will start an A/D conversion when a capture or compare event occurs. It has no effect when clear.

ROT  Bit 1. In Capture, a “1” resets opposite base timer (not TB), while a “0” has no action. In Compare, a “1” selects opposite base timer (not TB) for reset, while a “0” selects the TB bit timer for reset.

ON/RT  Bit 0. On Captures, a “1” means that the old data can be overwritten on an overrun, while a “0” means that the new data is lost (ignored) on overruns. In Compare, a “1” means that the timer selected by TB and ROT will be reset. A “0” will have no action.

The EPA(TIME register has two functions, depending on the mode of the channel. In Capture mode, the register is double buffered and holds the time of the transition (i.e., the value of the selected timer at the instant the transition was detected is saved in the EPA(TIME register). If the old time has not been read and the buffer is full when a new transition occurs, an overrun interrupt request will occur.

An interrupt is generated on the load of the EPA(TIME value, either from the buffer or directly if the buffer is empty. The EPA(TIME value must read each interrupt service in order to obtain more than one interrupt.

In Compare mode, EPA(TIME is programmed with the time that events are to occur. Multiple events can occur per time match. For example, the output pin can be made to set/reset/toggle, and generate internal functions such as starting the A/D and resetting a timer.

Note: The EPA and Compare CONTROLn registers should be written as words.
7.3 EPA Interrupts

There are 24 sources of interrupts within the EPA; 12 event interrupts (10 Capture/Compare and 2 compare only), 10 input overflow interrupts (for overrun errors) and two timer overflow interrupts.

Interrupts can be masked either in the core (for events on channels 0-3), or in the EPA mask registers for all other sources. See Figure 7-3 for bit map of EPA_MASK, and EPA__MASK1. All channels excluding EPA0-3 share a common interrupt request line to the core, via EPAINTx. This means that EPA channels 0-3 can have their interrupt requests serviced directly, while all other sources must be decoded.

The decoding is handled by the use of the EPAIPV (EPA Interrupt Priority Vector) which is located at 1FA8h. This register always contains a code for the next highest priority interrupt which is both pending and masked. All EPAIPV values are shown in Figure 7-4.

This register is designed to be used in conjunction with the TIIJMP instruction to vector to the appropriate interrupt service routine. The EPAIPV should be read until it returns 00h before exiting the EPAINTx service routine. This insures that all pending and masked interrupts have been acted upon; also this is the only way to clear the EPAINTx pending bit in the INT_PEND and pending bits in the EPA__PEND / EPA__PEND1 registers.

When using the EPAIPV with the TIIJMP instruction, some care must be used. The EPAIPV register always returns a number with the LSB clear. (ie 02,04 ...). However, the TIIJMP multiplies this by two when calculating the offset into the jump table. The result of this is that consecutive jump vectors will not be consecutive in memory. There will always be an unused word between them.

See the program example 16 in the EPA outputs section for details on using EPAIPV with TIIJMP.

The EPA__MASK and EPA__MASK1 registers are only word addressable. Do NOT attempt to write to them as bytes as this will have no effect.

![EPA IPV Register](image)

**Figure 7-4. EPA Interrupt Priority Vector**

**Table 7-3. The EPA_PEND and EPA_MASK Registers**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPA</td>
<td>EPA</td>
<td>EPA</td>
<td>EPA</td>
<td>EPA</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
</tr>
</tbody>
</table>

**EPA_PEND1 (1FA6H: Byte), EPA__MASK1 (1FA4H: Byte)**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsv</td>
<td>rsv</td>
<td>rsv</td>
<td>COMP</td>
<td>COMP</td>
<td>OVR</td>
<td>OVR</td>
<td>OVR</td>
</tr>
</tbody>
</table>

**Figure 7-3. The EPA_PEND and EPA_MASK Registers**
7.4 Input Capture

The Capture modules of the EPA can be used, among other things, to time-stamp events which occur on EPA input channels. When an event occurs, the value of the selected timer is loaded into the EPA-TIMEn registers associated with the channel on which it occurred. If the EPA-TIMEn register is full, then the data is buffered. However, if the buffer is also full, then the data in the buffer will be lost, depending on the state of the ON/RT bit in the EPA-CONTROLn register.

An interrupt is generated each time the EPA-TIMEn register is loaded; whether directly or from the buffer. Therefore, the EPA-TIMEn register must be read even if the data is not being used to allow another interrupt to be generated.

7.4.1 HSI EXAMPLE #1

Consider the problem of synchronizing an A/D conversion with a clock pulse. This can be done using an EPA capture channel which is programmed to start an A/D conversion. (See Program 9).

First, the A/D is programmed to perform a conversion on channel 3. The conversion is programmed to be started by the EPA. Next Port baud 1.0 is set up for use by the EPA, and EPA0 is programmed to look for a rising edge.

When the EPA senses a rising edge on channel 0, it stores the value in TIMER1 into EPA-TIMEn, and starts the A/D converter.
7.4.2 HSI EXAMPLE #2 : ABS

One problem in implementing an Anti-Lock Braking System is determining the individual speed of the wheels. The following program 10a shows one way (using both the EPA and PTS) to calculate the speed of a wheel. It is easy to modify the program to perform speed calculations for 4 wheels. Just add in three more EPA channels, and PTS control blocks.

It is assumed that a square wave is fed into an EPA channel whose frequency is proportional to the speed of the wheel. EPA channel 0 is used to look for rising edges. The number of rising edges seen in a five millisecond interval is counted, and this is used to determine the average wheel speed.

The PTS is used to count the number of pulses received, and to determine the first and last time that a rising edge is detected during a five millisecond loop.

Figure 7-5 shows a sample input to the EPA. The following sequence of events takes place to determine the wheel speed:

1. The first edge in the 5ms interval causes a normal EPA interrupt. The time from EPA_TIME0 is saved into ITIME. The PTS is enabled to handle counting the rest of the edges.
2. Edges 2 through “n” cause a PTS cycle to occur. The PTS moves EPA_TIME0 to FTIME and decrements PTSCOUNT. If the PTSCOUNT is initialized to 0FFh, negating PTSCOUNT will yield the number of edges captured by the PTS plus the first edge seen by the normal interrupt for edge 1.
3. The 5ms interrupt disables the PTS channel (so the next rising edge will cause a normal software interrupt repeating the cycle). It also calculates the wheel speed using the following formula:

\[
\text{Speed} = \frac{(\text{FTIME}-\text{ITIME}) \times \text{conv\_fact}}{\text{NEG(PTSCOUNT)}}
\]
Program 10a. ABS Input Frequency Detection using the PTS and EPA Inputs

```asm
\text{Include(Kr-inc)}
\text{; SFR's accessed by window 1F} \text{; }
\text{PT10_1F} \text{ EQU 0D2H:BYTE ; R/W Window 1FH (1FD2H)}
\text{PTState_1F} \text{ EQU 0DXH:BYTE ; R/W Window 1FH (1FDXH)}
\text{Comp_Time_1F} \text{ EQU 0AEH:WORD ; R/W Window 1FH (1FAEH)}
\text{Comp_Control_0_1F} \text{ EQU 00RH:BYTE ; R/W Window 1FH (1FMRH)}
\text{EPA_Mask_1_1F} \text{ EQU 0A1H:BYTE ; R/W Window 1FH (1FA1H)}
\text{Timer1_Control_1F} \text{ EQU 098H:BYTE ; R/W Window 1FH (1F98H)}
\text{; } \text{program equates} \text{; }
\text{Speed_high_Constant} \text{ EQU 007H ; this is our magic number}
\text{Speed_low_Constant} \text{ EQU 0A120H ; in order to get Hz in result using 16MHz}
\text{; } \text{general purpose RAM} \text{; }
\text{reg at 080h}
\text{; PTS Control Block for EPA0}
\text{PTS_Count_EPA0} \text{ DSB 1}
\text{PTS_Control_EPA0} \text{ DSB 1}
\text{PTS_Source_EPA0} \text{ DSW 1}
\text{PTS_Delay_EPA0} \text{ DSW 1}
\text{FTIME_0} \text{ DSW 1 ; Final Time in 5ms window}
\text{num_of_pulse_0} \text{ DSW 1 ; number of pulses}
\text{inv_speed_0} \text{ DSL 1 ; speed indicator}
\text{Templ} \text{ DSL 1 ; Long temp register}
\text{EPA_TVP_Ptr} \text{ DSW 1 ; Priority Vector Shadow Reg for TlagMP}
\text{EPA_INTX_TBase_Ptr} \text{ DSW 1 ; EPAx Jump Table Base pointer}
\text{; } \text{Interrupt Vectors} \text{; }
\text{casr at 200H}
\text{DCW EPA0_ISR}
\text{casr at 204H}
\text{DCW PTS_COUNT_EPA0}
\text{casr at 2000H}
\text{DCW EPAx_ISR}
```

270873-41
Program 10b. ABS Input Frequency Detection using the PTS and EPA Inputs

; main routine

; oeeg at 208h

Initial_Start:
  DI          ; Disable interrupts
  DPI         ; Disable PTS
  LD          ; Initialize Stack
  Clear_RAM

Software_Tmr_Init:
  LDB  WSR, #1Fh    ; Use Window 1FH (1F00-1FFFF -> 80-FF)
  LD  EPA_MASK1, 1Fh, 08h ; Enable compare module interrupt
  ORB  Int_Mask, #1Fh
  LDB  Comp_Control0, 1Fh, #40h ; Enable EPA0 & EPAAX interrupts
  LDB  Comp_Tim0, 1Fh, #2500 ; Initialize periodic 5ms interrupts
  CLRB  WSR    ; Reset WSR to zero window
  LD  EPAINTX_JSBase_Ptr, #EPAINTX_JSBase ; TI JMP Table Pointer

PTS_Init:
  LDB  PTS_Ctrl, EPA0, #08h ; Single word transfer
  LD  PTS_SRC, EPA0, #EPA_TIME0
  LD  PTS_DEST, EPA0, #F_TIME_0

EPA_Init:
  LD  TempL, #0FE20H ; CAPTURE ON POSITIVE EDGES
  ST  TempL, #EPA_CONTROL0

Port_Config:
  LDB  WSR, #1Fh    ; Use Window 1FH (1F00-1FFFF -> 80-FF)
  ORB  P1S_SEL, 1F, #01h ; EPA0 Configure
  ORB  P1O, 1F, #01h ; Set as FSI
  LDB  TIM0_INT_CONTROL, 1F, #0C0h ; Turn on TIMER1
  CLRB  WSR ; Reset WSR to 0

EPTS ; Enable interrupts

Loop_Forever:
  JMP Loop_Forever ; Let interrupts take-over

; Foreground processing can use the wheel speed indicators. The processor
; will be interrupted every five milliseconds for wheel speed calculations.

Clear_RAM:
  CLR  ITIME_0  ; Clear initial time values
  CLR  FTIME_0  ; Clear final time values
  RET

; EPA INTERRUPT SERVICE

EPA0_ISR:
  PUSHA
  LDB  PTS_SELECT4, #05h ; check for int bug
  LD  TIME_0, EPA_TIME0 ; store initial time
  LDB  PTS_COUNT, EPA0, #FFFFH ; set count = 255
  ORB  PTS_SELECT, #10h ; Next interrupt will be PTS
  POPA
  RET

FIX2:
  ORB  INT_PEND, #10h ; force call to PTS
  POPA
  RET

270873-42
Program 10c. ABS Input Frequency Detection using the PTS and EPA Inputs

```assembly
; Software Timer Interrupt

EPAx_ISR:
PUSH A
  ; Save PSW
LD A, EPAIPv_Ptr(#EPA_PRIORITY)
  ; Read EPAIPv
TIMP EPAIPv_Ptr(#EPAIPv_Ptr)8H

Sw_Timer_ISR:
  AND WSR
  ; PTS Select #OFF1h
  ; Turn off PTS EPA interrupts
LD B WSR#1H
ADD COMP_TIME0.1F#2500
  ; Re-calculate COMP0 and reset
CLR C WSR
CLR D INV_SPEED0+2
  ; Clear Wheel Speed Registers
CLR E num_of_pulse0+1
  ; Clear pulse counter for new

Wheel_speed_Calc_0:
  NEGB PTS_COUNT_EPA0
  ; Negate the Pulse Count
  ; Speed high Constant
SUB F INV_SPEED0/TIME0/NUM0/PULSES0
  ; get total period
DIV G INV_SPEED0/NUM0/PULSES0
  ; average Timer1 tick count
LD H Tempi2+2
  ; Load Tempi with MAGIC number
LD I Tempi2+2
  ; Speed low Constant
DIV J INV_SPEED0
  ; to give units in Hertz
ST K INV_SPEED0
  ; Result of wheel speed in Hertz
ST L INV_SPEED0
  ; stored in INV_SPEED0

Call Clear_RAM
LD A, EPAIPv_Ptr(#EPA_PRIORITY)
  ; Read EPAIPv
TIMP EPAIPv_Ptr(#EPAIPv_Ptr)8H

EPAxDone
  ; POPA
  ; RET

EPAxJBase
  ; DCW EPAxDone
  ; DCW Error_Loop
  ; DCW Error_Loop
  ; DCW Error_Loop
  ; DCW Error_Loop
  ; DCW Error_Loop
  ; DCW Error_Loop
  ; DCW Sw_timer_ISR

Error_Loop:
  TIMP ERROR_LOOP
```

270873-43
7.5 EPA HSO Generation

Control over the generation of HSO, High Speed Output, is gained by the use of two SFRs, EPA__CONTROLn and EPA__TIMEEn ("n" designates the number 0-9, of the EPA channel) for each output. The EPA__CONTROLn register controls the nature of the action to be taken when the EPA__TIMEEn register matches the value in the specified time base register (i.e., either TIMER1 or TIMER2).

The event must be programmed by first writing to the EPA__CONTROLn register, and then to the EPA__TIMEEn register. If the RE bit in the EPA__CONTROLn register is set, then the event programmed will occur every time a match between the time base and EPA__TIMEEn occurs; otherwise the event will be disabled after the first match occurrence, and can be re-enabled by writing to the EPA__TIMEEn register.

Note, Port1 must be configured for use by the EPA before any HSO can be generated. This is accomplished by writing a 1 to P1REG.n to turn off the pull-down, writing a 0 to the P1IO.n register to configure the pin as an output, and lastly, writing a 1 to P1SEL.n bit to select P1.n for use by the EPA.

An interrupt will be generated each time a match occurs on an enabled channel (i.e., one where EPA__CONTROL was written). Depending on the channel on which the interrupt occurs, it can be masked in one of two places: either in the core for channels 0-3, or in the EPA mask registers and the core EPAINTx bit for all other channels. See the EPA interrupt section for details.

7.5.1 SQUARE WAVE GENERATION

To generate a simple square wave output, the following code can be used. It first configures pin P1.1 for use with the EPA as a push/pull output. Next EPA channel 1 is configured to toggle its output, with automatic event re-enable, every time the value in Timer1 matches EPA__TIME1. EPA__TIME1 is initialized to be 3000h and Timer1 is set-up. Now, the EPA will automatically toggle EPA1 every time the value in Timer1 reaches 3000h.

A square wave with a duty cycle of other than 50% can also be generated by using two channels in conjunction (EPA0 and EPA1, EPA2 and EPA3, COMP0 and EPA8, or COMP1 and EPA9). The following code generates two square waves, one with a 30% duty cycle, and the other with a 60% duty cycle. Both square waves have the same frequency.

First, the program configures Port 6 pins 0 and 1 for use by the EPA. Then the EPA channels are set up. For the 30% duty cycle wave, EPA8 is programmed to clear the output pin after 100 counts of TIMER1. COMP0 is programmed to set the output pin after 300 counts, and then reset the timer. Both channels have the re-enable bit set so the event will repeat automatically.

The 60% duty cycle is produced in a similar way, except that COMP1 is not programmed to reset the timer as this is being handled by COMP0.
Program 11. Generating 2 PWM Pulses Using No CPU Overhead

This Program demonstrates the use of the EPA Compare function to produce two PWM outputs. No CPU Overhead, No RTS.

```
EPA_CONT8 EQU 0E0h ; window to EPA_CONTROL8
EPA_TM8  EQU 0E2h  ; window to EPA_TM8
EPA_CONT9 EQU 0E4h  ; window to EPA_CONTROL9
EPA_TM9  EQU 0E6h  ; window to EPA_TM9
COMP_CONT0 EQU 0E8h  ; window to COMP_CONTROL0
COMP_TM0  EQU 0EAh  ; window to COMP_TM0
COMP_CONT1 EQU 0ECah ; window to COMP_CONTROL1
COMP_TM1  EQU 0EEh  ; window to COMP_TM1
TIMER1_CON EQU 0F8h  ; window to TIMER1_CONTROL
PREG_REG W EQU 0F5h  ; window to PREG
PSEL_W   EQU 0F1h  ; window to PSEL
PHIO_W   EQU 0F3h  ; window to PHIO

CSEG AT 20380h
```

- **LD#** WSR.?7ch ; window 150-1FPM to 80-FF
- **LD#** EPA_CONT8.?58h ; reset pin, event re-enable
- **LD#** EPA_CONT9.?58h ; reset pin, event re-enable
- **LD#** EPA_TM8.?100 ; 20% duty cycle
- **LD#** EPA_TM9.?200 ; 60% duty cycle
- **LD#** COMP_CONT0.?65h ; reset pin, event re-enable
- **LD#** COMP_CONT1.?65h ; reset pin, event re-enable
- **LD#** COMP_TM0.?100 ; same frequency as EPA8
- **LD#** COMP_TM1.?100 ; same frequency as EPA8
- **LD#** TIMER1_CON.?0C2h ; enable timer, count at 1us rate

Self: Br Self ; set EPA take over

---

Figure 7-6. Output Generated by Program 11
7.5.2 PWM SIGNAL GENERATION WITHOUT PTS

Up to four PWM outputs can be generated in a manner similar to the square wave generation shown on the previous page.

One way to change the duty cycle would be to write a routine which monitors the state of the output. When it goes low, EPA__TIME0 register could be changed. If EPA__TIME0 was written when the clock was high, it would be possible for the duty cycle to become 100% for one cycle. (If EPA__TIME0 was written to a value less than TIMER1.)

Two nice things about using this method to generate a PWM is that it doesn't require any CPU or PTS overhead to maintain. And, any frequency and duty cycle can be produced with 16-bit resolution for both.

However, two EPA channels are being used for each PWM signal, and one dedicated timer is needed.

It is possible to generate a PWM signal using only one channel. Thus, up to 10 slower PWM signals can be generated. The code below demonstrates the method. For simplicity, only one PWM is produced.

The method used is similar to the HSO/CAM method used on other MCS-96 devices, such as the 8XC196KB. A wide range of duty cycles and frequencies can be produced. Only one timer is used.

The first segment of code sets up the registers used to hold the high (Const1) and low (Const2) time values of the PWM output. The next section configured PORT1.0 to be used by the EPA as an output. Following this, the EPA__CONTROL0 register is programmed. It is set up to toggle the output pin every time the value of TIMER1 matches EPA__TIME0.

The RE bit should NOT be set as the EPA__TIME0 register will be re-written after each edge, enabling the next event. The last part of the main loop starts TIMER1 running at 1us per count, and enables the EPA0 interrupt in the core.

The interrupt service routine is requested each time the EPA__TIME0 matches TIMER1. It checks the state of the PWM output (EPA0), to determine which value to add to the EPA__TIME0 register to set up for the next edge.

For example if the output is high, the value of Const1 is added to EPA__TIME0 and stored back into EPA__TIME0. The interrupt service routine will take 90 state times to set up a rising edge, plus an additional seven if the duty cycle has to be changed, and 79 states to set up a falling edge.

The duty cycle can be changed by modifying the values of Const1 and Const2. However, their sum must be kept the same to avoid changing the frequency. To change the duty cycle using the above code, the new values for the constants should be written to NConst1 and NConst2, followed by the setting of the “valid” flag.

In the previous example, it is assumed that the frequency is measured from rising edge to rising edge. So the values of the constants are changed by the interrupt service routine only after a rising edge was set up (i.e., the output is currently low). This insures that the frequency will not change momentarily.
Program 12. PWM Generation Using Interrupts

; This program demonstrates the use of the EPA Compare function to
; produce a PWM output. The PTS is not used.

; EPA and SFR window definition for window 7B
EPA_COMPARE0_W   EQU 020h:BYTE
EPA_COMPARE1_W   EQU 021h:BYTE
EPA_COMPARE0_W   EQU 022h:WORD
EPA_COMPARE1_W   EQU 023h:WORD
EPA_COMPARE0_W   EQU 024h:WORD
EPA_COMPARE1_W   EQU 025h:WORD

; port1 SFRs for window 7E
PSEG1_W   EQU 0F0h:BYTE
P10_W    EQU 0F1h:BYTE
PREG_W   EQU 0F2h:BYTE
P1_W     EQU 1F0h:BYTE ; P1 register
Const1   EQU 060h:WORD ; count for EPA0 high
Const2   EQU 062h:WORD ; count for EPA0 low
Temp    EQU 064h:BYTE ;
NConst1  EQU 066h:WORD ; new value for EPA0 high
NConst2  EQU 068h:WORD ; new value for EPA0 low
Valid   EQU 06Ah:BYTE ; new data is valid

CSEG AT 200h

CSEG AT 200h

DCCW EPA0_ISR

DI

DPTA

LD SP 500h ; set up stack
LD Const1 #100 ; 20% duty cycle
LD Const2 #400 ; 2KHz freq.
LD B WSR #7Eh
OR B PREG_W #01h ; set P1 high
AND B P10_W #0Fh ; make P1.0 output
OR B PSEG1_W #01h ; make P1.0 for EPA

LD B WSR #7Bh
LD EPA_COMPARE0_W #070h ; Timer1, toggle output
LD EPA_COMPARE0_W,Const1
LD B temp #02h ; enable timer, 1us
SB temp, TIMER1,CONTROL ; start timer
OR B INT_MASK #10h ; Enable EPA0, interrupt
CB Valid ; no new data as of yet
EI

SEI SImp Self

; To change the duty cycle of the PWM output, change the values of Const1 and Const2, but keep the
; sum the same or the frequency will change. Then set VALID indicating new data for the PWM.

; Rising edge to rising edge defines the frequency, and so the data will only be changed after the
; rising edge set up. This ensures that the new frequency will not change, momentarily, as a result
; of changing the duty cycle.

EPA0_ISR:

PUSHA

LD B WSR #7Eh ; read p1.0 thru window
ADD temp,Const2, EPA_COMPARE0_W set up rising edge
JBV valid,notout ; if new data exist
LD Const1, NConst1 ; update const1
LD Const2, NConst2 ; update const2
CB Valid ; more data was updated
JSMP out

SETA Add temp,Const1, EPA_COMPARE0_W set up falling edge
OUT ST temp, EPA_COMPARE0_W
POP A
RET

270873-46
7.5.3 PWM GENERATION WITH PTS

The PTS has two modes which can be used to generate PWM signals: PWM (up to 2 PWMs) and PWM TOGGLE (up to 4 PWMs). The latter uses the same method as shown on the previous page, but the PTS instead of a ISR handles updating the EPA__TIME0 register.

Program 13 produces the same results, but uses the PTS in PWM TOGGLE mode. The first block of code initializes the PTS control block. The PWM toggle mode is used, with the source registering pointing to the EPA__TIME0 register. Const1 and Const2 are chosen so that a 2KHz waveform with a 20% duty cycle is produced.

PORT1 is then configured, EPA channel 0 is programmed, and TIMER1 is started. It is important to note that EPA0 is forced low. This insures that the proper polarity PWM is generated. The EPA will just toggle the output, not caring what the initial state was.

EPAINT0 is enabled in both the core and the PTS__SELECT register. This allows the PTS to continuously set up the edges, instead of having the CPU handle it.

However, due to a bug in the interrupt handler on A-Step silicon, an interrupt service routine for EPAINT0 is still needed. If a PTS interrupt should occur within the latency time of the starting of another normal interrupt, the PTS interrupt will NOT be serviced, but rather a call will be made to the interrupt routine corresponding to the PTS interrupt. Therefore, the interrupt service routine for EPAINT0 should set the interrupt pending bit for EPAINT0 and exit. This will force a call to the PTS service routine.

Note that the PTS routine only takes 15 states to execute; this is more than an 80% reduction in the time needed to maintain a PWM output.

The PWM Toggle PTS cycle operates as follows:
1) The value pointed to by PTS__SOURCE is read.
2) Const1 or Const2 is added to this value, depending on the state of TBIT in PTS__CONTROL. TBIT=0 selects Const1
3) The result is stored back into the value pointed to by PTS__SOURCE. The TBIT is then toggled.

The duty cycle can be changed in the program by first writing the new values for PTS__CONST1 and PTS__CONST2 into NCONST1 and NCONST2, respectively. Then the PTS should be disabled from servicing EPA0 interrupts.

The EPA0 ISR will change the duty cycle. If the output pin is in the high state when it is called, it will perform a "manual" PTS cycle to set up the falling edge. When it is called and the output is in a low state, it will update the PTS constants and re-enable the PTS for EPA0. Lastly, it will force a call to the PTS by setting the INT__PEND bit for EPA0.

Program 14’s methods work well for generating PWM output as long as there is enough time between edges for either the CPU or the PTS to set up the next edge. However, if two edges are placed very close together, there will not be enough time to set up the second edge.

---

**Figure 7-7. Output of Program 12 and 13**

---

270873–15
Program 13. Generate a PWM on EPA0 using the PTS Toggle Mode

This program demonstrates the use of the EPA Compare function with the PTS toggle mode to produce a PWM output.

EPA and SFR window definition for window 7E:

EPA_CONTROL0_W  EQU  0Eh.BYTE
EPA_TIME0_W  EQU  0E2h.WORD

Port1 SFRs used with window 7E:

PTSSEL_W  EQU  0F7h.BYTE
PT10_W  EQU  0F7h.BYTE
PT1REG_W  EQU  0F4h.BYTE
PTPIN_W  EQU  0F6h.BYTE

DCW  PTS_UNUSED  .PTS interrupt for EPA0
CSEG  AT  2080h

DCW  EPA0_ISR  .vector for EPA0 interrupt routine
RSEG  AT  0000h

PTS_UNUSED:  DSB  1
PTS_CON:  DSB  1
PTS_SRC:  DSB  1
PTS_CONST1:  DSB  1
PTS_CONST2:  DSB  1
NConst1:  DSB  1
NConst2:  DSB  1
CSEG  AT  2080h

START:  DI

DPTS
Ld  SP,#500h  .set up stack
Ldb  PTS_CON,#43h  .PWM toggle mode, CONST1 first
Ld  PTS_SRC,#EPA_TIME0  .source is EPA time0
Ld  PTS_CONST1,#100  .on for 100 us
Ld  PTS_CONST2,#400  .off for 400 us
LdB  WSR,#7Eh
OrB  PT1REG_W,#01h  .force P1.0 high
OrB  PTSSEL_W,#01h  .make P1.0 for EPA
AndB  PT10_W,#0F0h  .make P1.0 output
LdB  WSR,#7Bh
Ld  EPA_CONTROL0_W,#070h  .Timer1, toggle output
Ld  EPA_TIME0_W,PTS_CONST1
LdB  #0C2h  .enable timer, 1ms
SxB  #0h,TIMER1_CONTROL  .start timer
OrB  INT MASK,#010h  .Enable EPA0 interrupts
OrB  PTS_SELECT,#10h  .Enable EPA0 to PTS interrupts
De
EI  Self  Self

;To change the duty cycle, load the new values of the constants into NConst1 and NConst2. Then disable PTS servicing of the EPA0 interrupts. The EPA_ISR will handle the task of changing the duty cycle. If needed (if the output is high) it will first perform a manual PTS cycle.

EPA0_ISR:

PUSHA
JSR  PTS_SELECT,Abort  .check for bug
LdB  WSR,#7Eh
JSR  PTS_PIN,W,change
Ld  PTS_CONST1,NConst1  .change duty cycle
Ld  PTS_CONST2,NConst2
LdB  PTS_CON,#40h  .set TBIT to proper value
OrB  PTS_SELECT,#10h  .restart PTS
abort:  OrB  INT_PEND,#10h  .go to the correct place
POP A
RTS

change:  LdB  WSR,#7Bh  .set up falling edge
Add  EPA_TIME0_W,PTS_CONST1  .do manual PTS cycle
POP A
RTS

Program 270873-47
Program 14. Generate a PWM Using the PTS PWM Mode and the Re-Map Feature
The PWM mode of the PTS can be used to work around this. Two EPA channels are used together (either EPA0 and EPA1 or EPA2 and EPA3). (See Program 14) One channel will control the rising edge, and the other will control the falling edge of the output. Thus, there is no need for the PTS or the CPU to intervene between the edges, allowing them to be placed closer together. However, there must be time to set up two edges before another edge can occur. Program 14 demonstrates this mode with the PTS.

The code is similar to what was presented above, but there are a few differences to be pointed out. First, there are two PTS control blocks. One controls the time from rising edge to rising edge, and the other controls the time from falling edge to falling edge. Normally, these should be the same.

The frequency of the PWM wave is controlled by the constants in the two PTS control blocks, and the frequency of TIMER1. The duty cycle is controlled by the difference of the two time registers, EPA_TIME0 and EPA_TIME1. For this example the pulse will be high for 50μs.

The duty cycle can be changed in a manner similar to that which was used in the previous PTS example. Note, however, that only EPA1 needs to be disconnected from the PTS, as it controls the falling edge.

Another thing to note is the manner in which the two EPA channels are configured. EPA0 is set up to force EPA1 high when EPA_TIME0 matches TIMER1, while EPA1 is set up to clear EPA1 when EPA_TIME1 matches TIMER1. Also, the 8th bit in EPA_CONTROL1 must be set to a 1. This is to allow EPA_CONTROL0 to control EPA1.

A word of caution is needed here: do NOT set both EPA_TIME0 and EPA_TIME1 to the same value and expect to get a 0% duty cycle PWM. This will not happen due to the fact that when there is a conflict between EPA commands (i.e., set and clear the pin at the same time), the EPA will toggle the pin at the EPA_TIME value.

Finally, note that the PWM output appears at EPA1 (or EPA3 if EPA2 and EPA3 are working together). Thus, PORT1 pin 1 is configured as an output for the EPA. Pin 1.0 can still be used as an LSIO pin.

7.5.4 PWM GENERATION USING SOFTWARE

As a final example Program 15, creating an PWM output on channel 9, will be considered. Since only channels 0-3 have direct interrupt lines to the core and PTS, this example will also demonstrate the use of the TJMP instruction. The PTS cannot be used with channels 4-9 since there is only one bit in the PTS_SELECT register for all 5 channels. The PTS cannot determine which channel caused the interrupt, and is therefore unable to modify the proper EPA_TIME register.

NOTE: Parts of the jump vector table were left out. They all contain a jump to the error routine.

This program is very similar to the first PWM example, except that a different EPA channel is used. Since EPA9 is used, and it doesn't have a direct interrupt line to the CPU, the following changes have been made. First the EPAINT9 bit in the EPA_MASK register has been set, allowing EPA9 interrupts to occur. The EPAINTx flag in the INT_MASK has also been set.
Program 15a. Generate a PWM Output Using EPA9 and Software Interrupts

```
#include (Macro.KR)

; This program demonstrates the use of the EPA Compare function to produce a PWM output. The
; T1MP instruction is also demonstrated.

; EPA and SFR window definition for window 7C:
EQU 0E4h:BYTE
EQU 0E6h:WORD

; portl SFRs for window 7E
P8SEL_W EQU 0Ff1h:BYTE ; P8 register.
P80_W EQU 0Ff5h:BYTE ; count for EPA9 high.
Const1 EQU 066h:WORD ; count for EPA9 low.
temp EQU 068h:BYTE
EPAIPV_PTR EQU 075h:WORD ; pointer to EPAIPV SFR.
EPA_TI_BASE EQU 077h:WORD ; point to base of T1MP table.

CSEG AT 2000h
DCW EPAX_ISR ; vector for EPAX interrupt routine
CSEG AT 2080h

DI
DPTS
Ld 5P, #500h ; set up stack
Ld Const1, #100 ; 20% duty cycle
Ld Const2, #400 ; 2Khz freq.
LdB WSR, #7Eh
AndB P86_SEL, #0FDh ; make P8.1 output
OrB P86_SEL, #025h ; make P8.1 for EPA9
LdB WSR, #7Eh
Ld EPA_Control9_W, #070h ; Timer1 toggle output.
Ld EPA_TIME9_W, #000
Ld temp, #0C26h ; enable timer, 1us
SeB temp, #0C21 ; start timer
OrB INT_MASK, #01h ; Enable EPAX, interrupt
Ld EPAIPV_PTR, #EAh ; use EPAIPV_PTR as temp
Or EPAIPV_PTR, #0400h ; Enable EPA INT9
St EPAIPV_PTR, #EAh
EI

self: Jmp Self
EPAX_ISR:

PushA
Ld EPAIPV_PTR, #1FA8h ; set pointer to EPAIPV SFR
Ld EPA_TI_BASE, #1F51, #TABLE1 ; pointer to jump table
T1mp EPA_TI_BASE, EPAIPV_PTR, #0F7h ; other EPA ISR's go here

EPANYTH1:
Ldb WSR, #7Eh
JbS 0FS1h, self ; read p6.1 thru window
Add temp, #0E4h ; set up rising edge
Sjmp out

set:
Add temp, Const1, #0E4h ; set up falling edge
out:
St temp, EPA_TIME9_W
T1mp EPA_TI_BASE, EPAIPV_PTR, #0F7h ; handle next interrupt

EXIT1: POPA
RET ; Trap any other interrupt, non should have occurred.

error: Sjmp error
```

270873-49
Program 15b. Generate a PWM Output Using EPA9 and Software Interrupts

; Following is the table of entry points for the EPA ISRs. Note that there are actually two tables
; which are interleaved. This is because there is a small problem with the TIMP/EPAPIV. The
; EPAPIV increments in steps of two and the TIMP multiplies the offset by two. Thus the final offset
; into the table is always a multiple of 4 and not 2 as one would expect.

JUMP_TABLE1:
    DOW   EXIT1

JUMP_TABLE2:
    DOW   error ; EXIT1
    DOW   error ; Timer2_over1
    DOW   error ; Timer2_over2
    DOW   error ; OVRTINT_1
    DOW   error ; OVRTINT_2
    DOW   error ; OVRTINT_3
    DOW   error ; OVRTINT_4
    DOW   EPAINT9_1 ; EPAINT9_1
    DOW   EPAINT9_2 ; EPAINT9_2
    DOW   error ; EPAINT4_1
    DOW   error ; EPAINT4_2

END

270873-50
The interrupt service routine has also changed, since now the exact source of the interrupt must be determined in software. A TJMP instruction is used to minimize overhead. The TJMP table contains pointers to the various interrupt service routines. Actually, the table consists of two interleaved tables due to the way in which TJMP and the EPAIPV register work together. The EPAIPV vector always is a multiple of 2, and the TJMP instruction multiplies by 2 to calculate the offset into the table. Thus offsets into the jump table will always be a multiple of 4. (This may change on latter parts.)

The TJMP instruction takes three arguments: base location of the table, an offset into the table, and a mask for the offset. The mask for the offset can be used to change the interrupt priorities, but does nothing in the example. The EPAIPV (EPA Interrupt Priority Vector) always contains a value indicating the highest priority interrupt that is pending (and masked). This register should be read until it returns a 00h, indicating that all interrupts have been processed. This is the only way to clear the EPA__PEND and EPA__PEND1 register and INT__PEND bit 0 in the core. The above example does this by ending each EPA interrupt source's service routine with another TJMP instruction. When the EPAIPV returns a 00h, flow is handed over to the EXIT subroutine, which handles exiting from the EPAINTx service routine.

7.6 Top 5 Issues with the EPA

1. Read the EPA__TIME register after each EPA (input capture) interrupt.
2. All EPA__CONTROL and COMP__CONTROL registers should be written as WORD. This will make the users code compatible with future KX devices. Reserved bits are written to zero.
3. PWM Generation can be accomplished via:
   1. 10 PWMs with Software Interrupts (EPA0–EPA9)
   2. 4 PWMs with PTS PWM Toggle Mode (EPA0–EPA3).
   3. 2 PWMs with PTS PWM Mode and re-mapping outputs (EPA1 and EPA3)
   4. 4 PWMs using a dedicated Timer with Re-mapping (EPA1, EPA3, EPA8 and EPA9).
4. Before exiting an EPAINTx interrupt service routine, the EPAIPV register MUST be read until it equals "00".
5. Due to a Bug in A-step Silicon, the EPA__MASK1 and EPA__PEND1 must be written as words.