Keywords: MAXQ,microcontroller,current loop,process-monitoring

APPLICATION NOTE 3653

MAXQ Microcontroller Drives a Smart and Intelligent 4-20mA Transmitter

By: Franco Contadini
Dec 22, 2005

Abstract: The 4-20mA current loop is a common technique for transmitting sensor information in industrial process-monitoring applications. (Sensors measure physical parameters such as temperature, pressure, speed, and liquid flowrates.) Current-loop signals are relatively insensitive to noise, and their power can be derived from a remotely supplied voltage. This makes current loops particularly useful when the information must travel a long distance to a remote location.

Straightforward Loop Operation

In a current loop, the output voltage from a sensor is first converted to a proportional current, in which 4mA normally represents the sensor's zero-level output and 20mA represents the full-scale output. A receiver at the remote end converts the 4-20mA current back to a voltage, which can be further processed by a computer or display module.

The typical 4-20mA current-loop circuit consists of four elements: a sensor/transducer, a voltage-to-current converter, a loop power supply, and a receiver/monitor. In loop-powered applications, the sensor drives the voltage-to-current converter, and the other three elements are connected in series to form a closed loop (**Figure 1**).

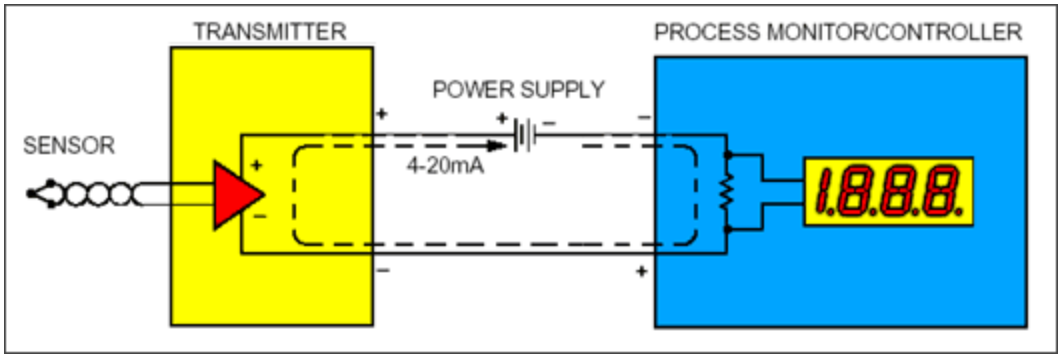

Figure 1. Diagram of a 4-20mA loop-powered circuit.

The Smart 4-20mA Transmitter

Traditionally, a 4-20mA transmitter included a field-mounted device that sensed a physical parameter and generated a proportional current in the standard range of 4-20mA. Responding to industry demand,

the second-generation 4-20mA transmitters, called 'smart transmitters', use a microcontroller (µC) and data converter to condition the signal remotely.

Smart transmitters can normalize gain and offset, linearize the sensor by converting its analog signal to digital (RTD sensors and thermocouples, for example), process the signals with arithmetic algorithms resident in the µC, convert back to analog, and transmit the result as a standard current along the loop.

The newest third-generation 4-20mA transmitters (**Figure 2**) are considered 'smart *and* intelligent'. They add digital communications which share the twisted-pair line with the 4-20mA signal. The resulting communication channel can transmit control and diagnostic signals along with the sensor data.

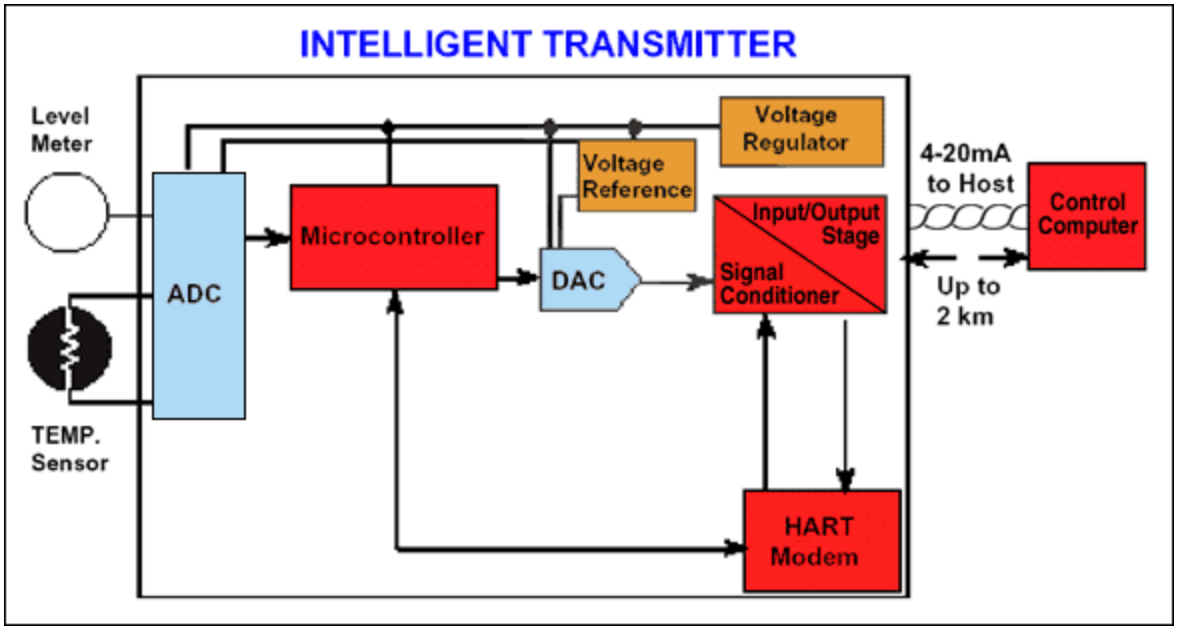

Figure 2. Diagram of a smart and intelligent 4-20mA transmitter.

The communication standard used by smart transmitters is the Hart protocol, which employs frequency shift keying (FSK) and is based on the Bell 202 telephone communication standard. Bits 1 and 0 of the digital signal are represented by the frequencies 1200Hz and 2200Hz, respectively. Sine waves at these frequencies are superimposed on the sensor's DC analog signal to provide simultaneous analog and digital communications (**Figure 3**).

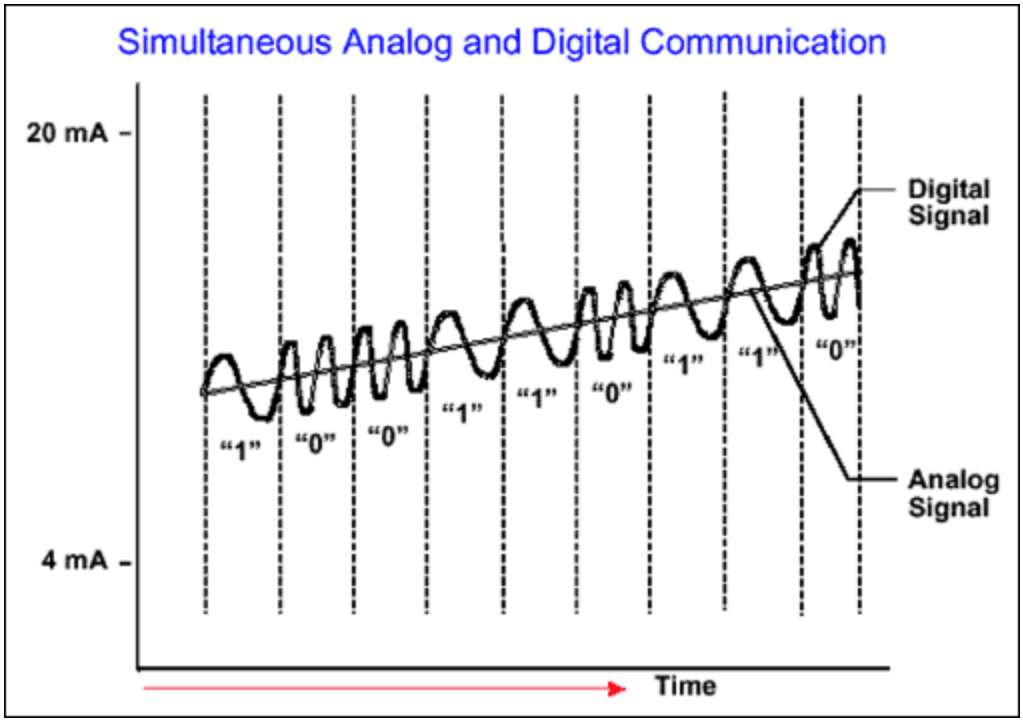

Figure 3. Simultaneous analog and digital communications.

The 4-20mA analog signal is not affected in this process because the average value of the FSK signal is always zero. The digital states can change two to three times per second without interrupting the analog signal. The minimum allowed loop impedance is 23Ω.

Basic µC Requisites for a Smart, Intelligent 4-20mA Transmitter

There are three specific capabilities that a µC must have to perform this 4-20mA current-loop application. The µC needs:

1. A serial interface to drive the ADC for data acquisition and the DAC for setting loop current.
2. Low power consumption, as the current budget is 4mA.
3. A multiply-accumulate unit (MAC), which both implements a digital filter applied to the input signal and also encodes and decodes the two frequencies of the Hart Protocol.

Selecting the µC

The above requisite capabilities are all available in the MAXQ family of RISC µCs (**Figure 4**).

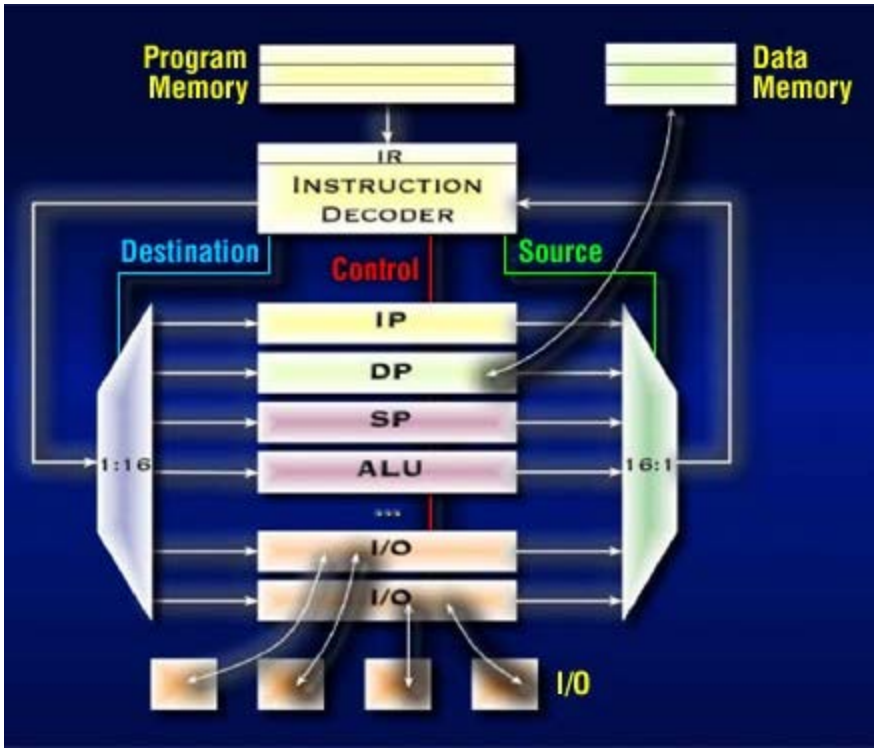

Figure 4. Illustration of the MAXQ µC architecture.

1. **Analog Functions**
   The MAXQ µC implements several analog functions. A clock-management scheme provides a clock *only* to those blocks currently in use. If, for example, an instruction involves the Data Pointer (DP) and Arithmetic Logic Unit (ALU), the clock is applied to those two blocks only. This technology reduces power consumption and lowers switching noise.
2. **Low Power Consumption**
   The MAXQ µCs have advanced power-management features that minimize power consumption by dynamically matching µC processing speed to the performance level required. Power consumption is slower, for example, during periods of reduced activity. To apply more processing power, the µC increases its operating frequency.

   Software-selectable clock-divide operations allow the flexibility of implementing a system clock cycle in 1, 2, 4, or 8 cycles of the oscillator. By performing this function in software, the µC enters a lower power state without the need, and cost, of additional hardware.

   Three additional low-power modes are available for extremely power-sensitive applications:
   - PMM1: divide-by-256 power-management mode
   - PMM2: 32kHz power-management mode (PMME = 1, where PMME is BIT 2 of the system clock-control register)
   - Stop mode (STOP = 1)

   In PMM1 mode, one system-clock cycle equals 256 oscillator cycles, which substantially reduces power consumption while the µC operates at reduced speed. PMM2 mode allows the device to run even slower, by using the 32kHz oscillator as a clock source. The optional switchback feature allows enabled interrupt sources like external interrupts, UARTs, and the SPI module to quickly exit the power-management modes and return to a faster internal clock rate. All these features give the MAXQ µC a 3MIPS/mA processing performance—far better than the closest alternative processor

(**Figure 5**).

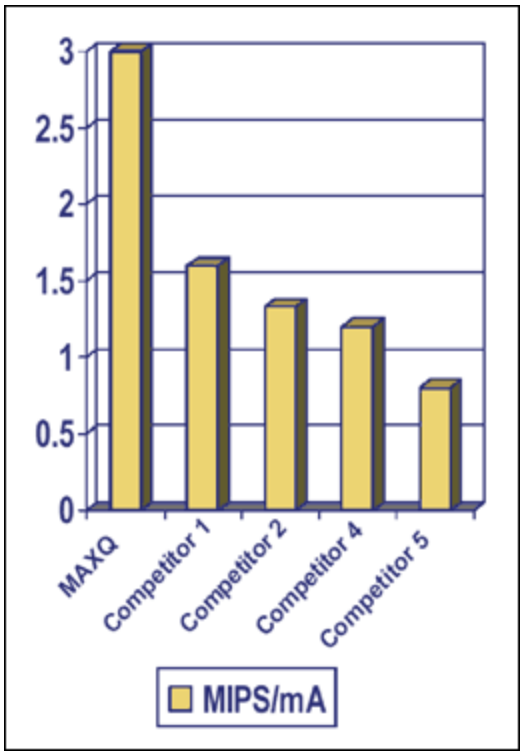

Figure 5. MAXQ performance in MIPS/mA is compared to competitive devices.

3. **Filtered Signal Processing**
   The MAC inside a MAXQ µC implements the signal processing required by a 4-20mA application. The analog signal is presented to an ADC, and the resulting sample stream is filtered in the digital domain. A general filter can be implemented using the following equation:

   $y[n] = \Sigma b_i x[n-i] + \Sigma a_i y[n-i]$

   where b_i and a_i characterize the system's feedforward and feedback responses, respectively. Depending on the values of a_i and b_i, a digital filter can be classified as a finite impulse response (FIR) or as an infinite impulse response (IIR). When the system contains no feedback elements (all $a_i = 0$), the filter is an FIR type:

   $y[n] = \Sigma b_i x[n-i]$

   If both the a_i and b_i elements are non-zero, however, the filter is an IIR type.

   As you can see from the equation above for an FIR filter, the main mathematical operation multiplies each input sample by a constant and then accumulates each of the products over the n values. Those operations are illustrated by the following C fragments:

```
y[n]=0;
for(i=0; i<n; i++)
y[n] += x[i] * b[i]
```

   The MAC of a MAXQ µC performs this operation with an execution time of 4 + 5n cycles, and with

a code space of only 9 words (vs. the 12 words required by a traditional µC and MAC).

```
move DP[0], #x                  ; DP[0] -> x[0]
move DP[1], #b                  ; DP[1] -> b[0]
move LC[0], #loop_cnt           ; LC[0] -> number of samples
move MCNT, #INIT_MAC            ; Initialize MAC unit

MAC_LOOP:

move DP[0], DP[0]                       ; Activate DP[0]
move MA, @DP[0]++                       ; Get sample into MAC
move DP[1], DP[1]                       ; Activate DP[1]
move MB, @DP[1]++                       ; Get coeff into MAC and multiply
djnz LC[0], MAC_LOOP.
```

(See the **Appendix** for details on the data memory access in the MAXQ architecture.)

Within the MAXQ's MAC, note that a requested operation occurs automatically when the second operand is loaded into the unit and the result is stored in the MC register. Note, also, that the MC register width (40 bits) can accumulate a large number of 32-bit multiply results before it overflows. That capability improves on the traditional approach, in which overflow must be tested after every atomic operation.

The Unique Capabilities of the MAXQ2000 µC

The first member of Maxim's MAXQ family is a low-power, 16-bit RISC microcontroller called the MAXQ2000. It incorporates an interface for liquid-crystal displays (LCDs) that drives up to 100 (-RBX) or 132 (-RAX) segments. While highly appropriate for blood-glucose monitoring, the MAXQ2000 is suitable for any application that requires high performance with low-power operation. It operates at a maximum of 14MHz (VDD > 1.8V) or 20MHz (VDD > 2.25V).

The MAXQ2000 includes 32kwords of flash memory (for prototyping and low-volume production), 1kword of RAM, three 16-bit timers, and one or two universal synchronous/asynchronous receiver/ transmitters (UARTs). For flexibility, separate supply voltages power the microcontroller core (1.8V) and the I/O subsystem. An ultra-low-power sleep mode makes the MAXQ2000 ideal for portable and battery-powered equipment.

MAXQ2000 Evaluation Kit

The powerful MAXQ2000 µC can be evaluated through its evaluation (EV) kit, a complete hardware development environment for the MAXQ2000 (**Figure 6**).

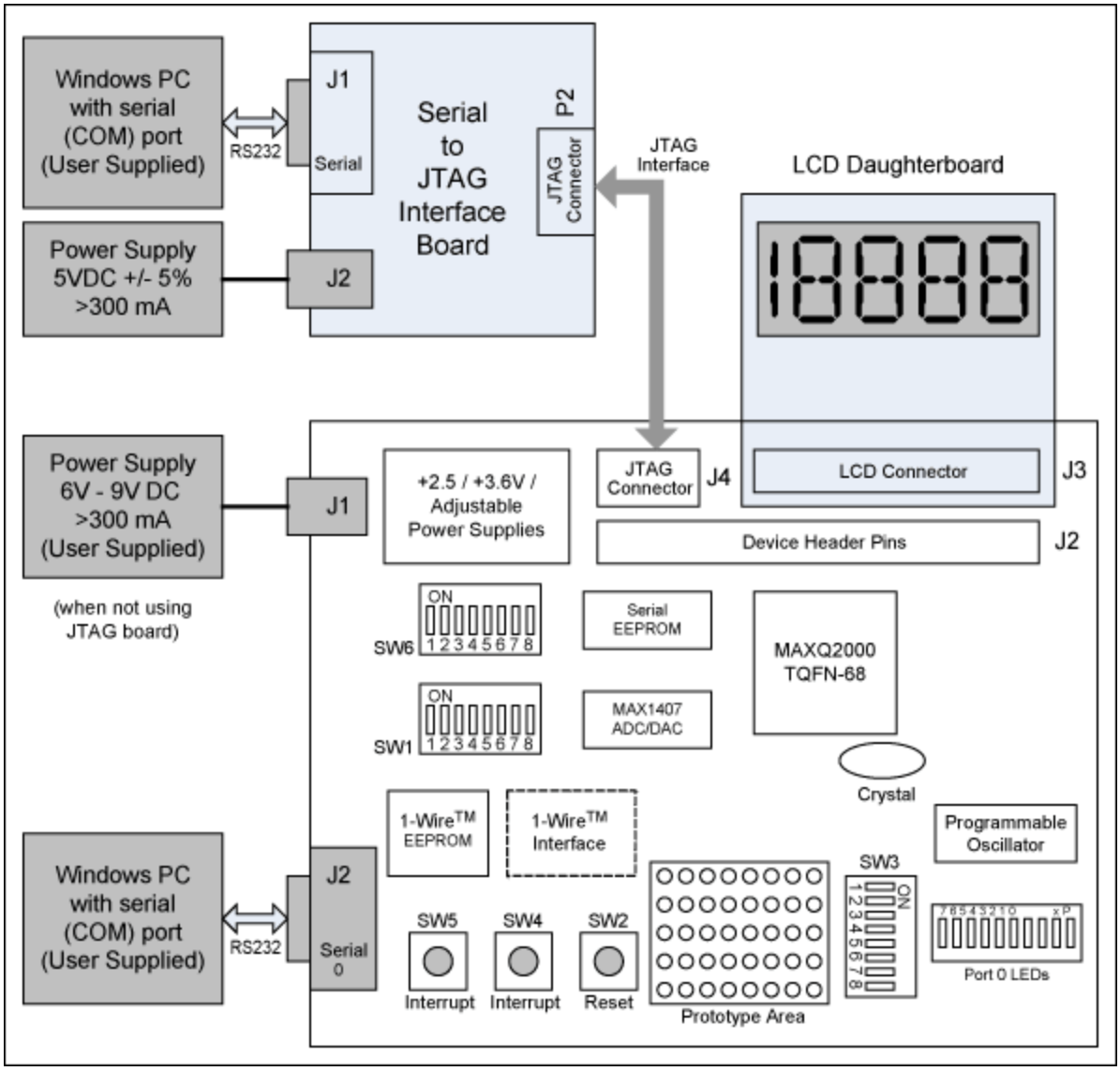

Figure 6. Block diagram for the MAXQ2000 EV kit.

The MAXQ2000 EV kit has the following characteristics:

- On-board power supplies for the MAXQ2000 core and VDDIO supply rails.
- Adjustable power supply (1.8V to 3.6V), which can be used for the VDDIO or VLCD supply rails.
- Header pins for all MAXQ2000 signals and supply voltages.
- Separate LCD daughterboard connector.
- LCD daughterboard with 3V, 3.5-digit static LCD display.
- Full RS-232 level drivers for serial UART (line 0), including flow control lines.
- Pushbuttons for external interrupts and microcontroller system reset.
- MAX1407 multipurpose ADC/DAC IC, connected to the MAXQ2000 SPI bus interface.
- 1-Wire interface and 1-Wire EEPROM IC.
- Bar-graph LED display for levels at port pins P0.7 to P0.0.
- JTAG interface for application load and in-system debugging.

Thus, the MAXQ2000 EV kit board has all the functions needed to implement a smart 4-20mA

transmitter: a low-power µC with true multiply-accumulate unit (for filtering and tone encode/decode); ADC for sensor reading; and a DAC for generating the analog output signal (**Figure 7**). With the addition of a low-power codec like the MAX1102, you can also implement a HART Modem.

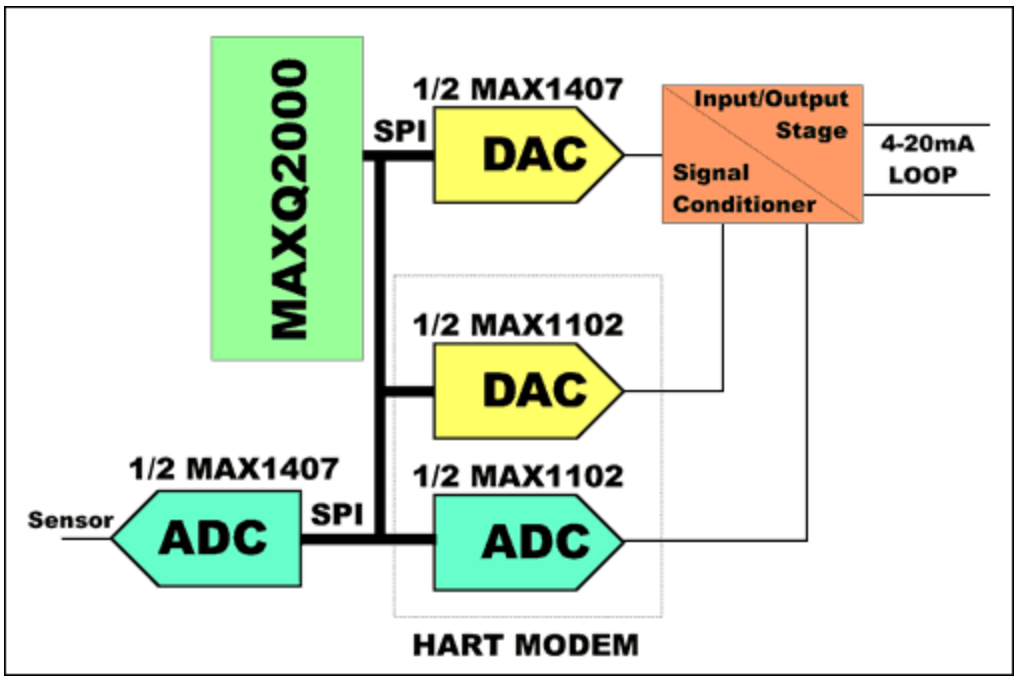

Figure 7. A 4-20mA transmitter based on the MAXQ2000 µC.

HART Modem Implementation

The MAC can be used to implement functions requested by a HART modem, if the system includes a tone encoder for 1200Hz and 2200Hz (representing bits 1 and 0), and tone detection for those frequencies.

To generate the required sinusoids, you can implement a recursive digital resonator as a two-pole filter described by the following difference equation:

$X_n = k * X_{n-1} - X_{n-2}$,

where the constant k equals 2 cos(2π*tone frequency/sampling rate). The two values of k can be precomputed and stored in ROM. For example, the value required to produce a tone at 1200Hz with 8kHz sample rate is k = 2 cos(2π*1200/8000).

You must also calculate the initial impulse required to make the oscillator begin running. If X_{n-1} and X_{n-2} are both zero, then every succeeding X_n will be zero. To start the oscillator, set X_{n-1} to zero and set X_{n-2} as follows:

$X_{n-2} = -A*\sin[2\pi(\text{tone frequency/sampling rate})]$

Assuming a unit sine wave for our example, this equation reduces to $X_{n-2} = -1\sin[(2\pi(1200/8000)]$. To further reduce it to code, first initialize the two intermediate variables (X1, X2). X1 is initialized to zero; X2 is loaded with the initial excitation value (calculated above) to start the oscillation. Thus, to generate one sample of the sinusoid, perform the following operation:

```
X0 = kX1 - X2
X2 = X1
X1 = X0
```

The calculation of each new sine value requires one multiplication and one subtraction. With a single-cycle hardware MAC on the MAXQ µC, the sine wave can be generated as follows:

```
move DP[0], #X1                         ; DP[0] -> X1
move MCNT, #INIT_MAC            ; Initialize MAC unit
move MA, #k                             ; MA = k
move MB, @DP[0]++                       ; MB = X1, MC=k*X1, point to X2
move MA, #-1                            ; MA = -1
move MB, @DP[0]--                       ; MB = X2, MC=k*X1-X2, point to X1
nop                                             ; wait for result
move @--DP[0], MC                       ; Store result at X0.
```

Because we only need to detect two frequencies, we use the modified Goertzel algorithm, which can be implemented as a simple second-order filter (**Figure 8**).

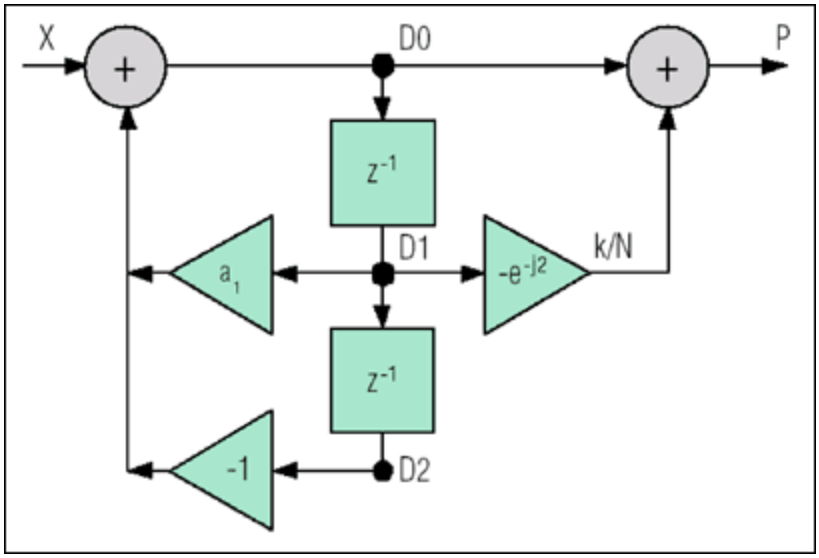

Figure 8. The Goertzel algorithm implemented as a simple second-order filter.

To use the Goertzel algorithm to detect a particular frequency, at compile time you first compute the value of a constant using the following formula:

```
k = tone frequency/sampling rate
a1 = 2cos(2πk)
```

Next, initialize the intermediate variables D0, D1, and D2 to zero, and perform the following for each sample X received:

```
D0 = X + a1*D1 - D2
D2 = D1
D1 = D0
```

After a sufficient number of samples has been received (usually 205 samples for an 8kHz sample rate), compute the following using the latest computed values of D1 and D2:

```
P = D12 + D22 - a1 * D1 * D2.
```

P now contains a measure of the squared power of the test frequency in the input signal.

To decode the two tones, we process each sample with two filters. Each filter has its own k value and its own set of intermediate variables. Each variable is 16 bits long, so the entire algorithm requires 48 bytes of intermediate storage.

Appendix. Data-Memory Access for the MAXQ Family

Data memory is accessed through either the data pointer registers DP[0] and DP[1], or the frame pointer BP[Offs]. When one of those registers is set to a location in data memory, that location can be read or written using the mnemonic @DP[0], @DP[1], or @BP[OFFS] as a source or destination.

```
move DP[0], #0000h                      ; set pointer to location 0000h
move A[0], @DP[0]                       ; read from data memory
move @DP[0], #55h                       ; write to data memory
```

Either of the data pointers can be post-incremented or post-decremented following a read. Alternatively, either data pointer can be pre-incremented or pre-decremented before a write access. Use the following syntax:

```
move A[0], @DP[0]++                     ; increment DP[0] after read
move @++DP[0], A[1]                     ; increment DP[0] before write
move A[5], @DP[1]--             ; decrement DP[1] after read
move @--DP[1], #00h             ; decrement DP[1] before write
```

Because the three pointers share a single read/write port on the data memory, the user must knowingly activate a desired pointer before using it for data-memory read operations. That can be done explicitly by using the data pointer select bits (SDPS1:0; DPC.1:0), or implicitly by writing to the DP[n], BP, or OFFS registers as shown below.

An indirect memory-write operation using a data pointer sets the SDPS bits, which, in turn, activates the write pointer as the active source pointer.

```
move DPC, #2            ; (explicit) selection of FP as the pointer
move DP[1], DP[1]       ; (implicit) selection of DP[1]; set SDPS1:0=01b
move OFFS, src  ; (implicit) selection of FP; set SDPS1=1
move @DP[0], src        ; (implicit) selection of DP[0]; set SDPS1:0=00b
```

Once a pointer selection has been made, it remains in effect until:

- The source data pointer-select bits are changed through the explicit or implicit methods described above (i.e., another data pointer is selected for use), or
- The memory to which the active source data pointer is addressing is enabled for code fetching by using the Instruction Pointer, or
- A memory write operation is performed using a data pointer other than the current active source pointer.

```
move DP[1], DP[1]               ; select DP[1] as the active pointer
move dst, @DP[1]                ; read from pointer
move @DP[1], src                ; write using a data pointer
                                        ; DP[0] is needed
move DP[0], DP[0]               ; select DP[0] as the active pointer
```

To simplify the data pointer increment/decrement operations without disturbing the register data, a virtual NUL destination has been assigned to system module 6, sub-index 7, to serve as a bit bucket. Data pointer increment/decrement operations can be done as follows without altering the contents of any other register:

```
move NUL, @DP[0]++              ; increment DP[0]
move NUL, @DP[0]--              ; decrement DP[0]
```

Related Parts		
MAXQ2000	Low-Power LCD Microcontroller	Free Samples
MAXQ2000-KIT	Evaluation Kit for the MAXQ2000	

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

Application Note 3653: http://www.maximintegrated.com/an3653
APPLICATION NOTE 3653, AN3653, AN 3653, APP3653, Appnote3653, Appnote 3653