INTRODUCTION

PID (proportional, integral, derivative) compensation is one of the most common forms of closed-loop control. Control of closed-loop systems that require compensation is a growing area of application for embedded microprocessors. In these systems, analog signals must be converted into discrete digital samples before compensation or filtering can take place. Loop performance dictates the sampling rate, and calculations must be complete before the next sample time begins. These loop-related constraints and the Nyquist frequency requirement place an upper bound on digital control of closed systems with feedback error. If the controlled system has a resonance or other behavior with a time constant shorter than the sample and calculation time, chaos is the most likely outcome. Despite these limitations, increases in microprocessor clock rates and the addition of on-chip control-oriented hardware are expanding the number of medium performance control applications handled by 8-bit machines. While an expensive DSP-class processor is the correct choice for the most demanding applications, several members of the M68HC11 family have the speed and resources to control multiple PWM channels.

This note provides two working examples of PID control-loop software. The first example, written primarily in C, shows a PID algorithm in a straightforward way using floating-point math. Key features of the C environment are covered for readers who are more used to assembly language. The second example implements a PID algorithm in assembly language. It uses the MC68HC11N4 on-chip math coprocessor to speed up arithmetic operations.

Both examples are complete and ready to run on a Motorola M68HC11EVS evaluation board. External interfacing is identical for both examples — an 8-bit analog to digital converter is used for input, and an 8-bit PWM waveform is output. Because the code in both examples carries more than 16 bits of precision, and because both processors support 16-bit PWM, only minor changes are needed to increase precision. Power amplifiers, sensors, and other interface circuitry must be supplied in order to experiment with real-world systems — a simple RC circuit is used for software checkout.


THE MICROCONTROLLERS

The MC68HC11K4 and MC68HC11N4 are 16-MHz devices with nonmultiplexed external address and data buses. Each has 24 Kbytes of on-chip ROM or EPROM. Both devices have multiple PWM channels with programmable period, duty cycle, polarity, and clock source. In both, two 8-bit channels can be concatenated to generate a 16-bit PWM output. The MC68HC11N4 also has two additional 12-bit PWM channels and two digital to analog converter channels with 8-bit resolution.
The MC68HC11N4 is particularly well-suited to PID computation because it has an on-chip math coprocessor that performs 16- and 32-bit multiplication and division, fractional division, and multiply-and-accumulate operations. All operations are done by means of memory-mapped registers, thus preserving the standard M68HC11 family instruction set. Multiplication and fractional division are complete in a maximum of 5 microseconds while integer division requires a maximum of 8.75 microseconds.

**PID ALGORITHM**

The general flow of a controlled system with PID compensation is shown in Figure 1. An informal review of the derivation of the discrete form of each term and how they function follows. REFERENCES 2 and 3 provide a complete analysis of digital PID control.

![Figure 1 PID Flow Diagram](image)

There is a desired setpoint in our process (Gd) and a measurement of the actual value G(t) in time. Error is:

\[ e(t) = G_d - G(t) \]

Output correction x(t) for the PID controller is:

\[ x(t) = K_P e(t) + K_I \int_0^t e(t) \, dt + K_D \frac{de(t)}{dt} \bigg|_{t = T} \]

where KP, KI, and KD are constants.

Now, rewriting the integral:

\[ x(t) = \left. K_P e(t) + K_I \int_0^t [G_d - G(t)] \, dt + K_D \frac{de(t)}{dt} \right|_{t = T} \]

To introduce discrete time, let \( t = kT \) where \( k = 1, 2, \ldots, n \) and \( T \) = the sampling and control update period. Now, \( t_0 = (k - 1)T \). The integral evaluated from \( (k - 1)T \) to \( kT \) can be approximated using the trapezoidal integration rule. The derivative of the error term is simply the rate of change of error, but this can be noisy over one period. Using a four-point central-weighted average for the difference term is a practical way to deal with this on a microprocessor.
The form which can be executed directly on the microprocessor is:

\[ x(t) = KP \cdot e(t) + KL \cdot (Gdt - \frac{T}{2} (G(Kt) + G((k-1)T))) + \frac{KD}{6T} \cdot ((e(kT) - e(k - 3) + 3(e(k - 1) - e(k - 2))) \]

This term is added to the current output and put into the PWM control register at the beginning of the next calculation cycle. Substituting the microcode labels for constants and variables into EQ. 4 and using C language operator notation gives:

\[ \text{NEWDTY} = KP \cdot (\text{ERRX}) + KI \cdot \text{PERDT} \cdot (\text{CMNDX} - (\text{ADRCX} + \text{ADRCXM1}) / 2) + (KD / (6 \cdot \text{PERDT})) \cdot ((\text{ERRX} - \text{ERRM3X}) + 3 \cdot (\text{ERRM1X} - \text{ERRM2X})) + \text{OLDDTY} \]

The function of the proportional term is clear, but the derivative and integral terms may need a brief explanation. When a system with only proportional control is off the specified setpoint, the controller will increase the control voltage until the error signal is zero, and the system thus returns to the setpoint with more applied voltage than is required for maintaining equilibrium. This causes overshoot and, as the process continues, under-damped ringing. The derivative term contributes proportionally to the error rate of change, but with the opposite sign of the proportional term. If the proper constants are chosen, critical damping can be achieved. The role of the integral term is to eliminate steady state error. A system that has a steady state error when tracking a ramping input function can use an integral term to integrate the error over time and compensate for it.

**C LANGUAGE IMPLEMENTATION**

This version of the PID control routine illustrates use of high-level language for control applications. High-level language offers many conveniences not available in assembly language. Here are a few instances:

- Memory-mapped registers can be defined in a single file, which can be included in any function.
- Since C is a strongly-typed language, the compiler can identify data-type mismatches, such as writing a 16-bit integer to an 8-bit port.
- Most C compilers for the M68HC11 family allow direct vectoring to interrupt service routines. Interrupt functions are usually defined for a special memory segment with a base address and a function name:

  ```c
  interrupt [INTVEC_START + 26] void RTI_interrupt(void);
  ```

  When an interrupt service routine is written it can be declared as an interrupt function instead of a normal subroutine:

  ```c
  interrupt void RTI_interrupt(void)
  ```

  The function is compiled with a terminating RTI instruction, eliminating the need for a patch between hardware interrupts and C subroutines. The void statements indicate that no arguments are being passed to or from the interrupt routine.

  One of the most attractive features of contemporary C compilers is the ability to add floating point math with an “include math.h” statement. Even when the final application can’t afford the time or code space for floating point calculations, use of floating point math during debugging provides an excellent means of testing new algorithms.

  Now to examine the control routine itself (refer to Appendix A for a complete C listing). After necessary files are included and floating point variables are declared, a prototype is given to define an assembly language function that is used later.
The main program initializes constants and variables, sets up the required on-chip peripherals, and waits for interrupts to occur. The M68HC11 real-time interrupt (RTI) is used to establish a precise time base for performing PID compensation. The period (T or PERDT) is determined by the RTI rate. In these examples, the period is 16.383 milliseconds, but this value is arbitrary — in real applications, the performance of the controlling microprocessor and the requirements of the controlled system determine the period.

The RTI interrupt function is a workhorse. It does PID loop PWM duty cycle calculations, performs I/O using the DOIO assembly language function, and checks the results against the usable PWM output range of $00 to $FF. If a floating-point result is out of range, the closest limit is substituted. This “saturation arithmetic” prevents out-of-range results from causing sign-reversals in the PWM output.

Details of error-checking and the DOIO subroutine are best understood by looking at the format of the floating-point variables. The four byte format is:

\[
\begin{align*}
&\text{S E E E E E E E E E} \\
&\text{E M M M M M M M M} \\
&\text{M M M M M M M M} \\
&\text{M M M M M M M M}
\end{align*}
\]

S represents a sign bit, E represents an 8-bit exponent biased by 127, and M represents a 23-bit fractional mantissa (significand) with an implicit leading 1. The I/O range of $00 to $FF is scaled into the eight most significant bits of a floating-point variable, giving a floating-point range of 1.0 ($3F800000) to 1.998046875 ($3FFF8000). The following expression is used to evaluate a floating-point number:

\[
F = \left( -1 \right)^S \times 2^{(E-127)} \times [1.M]
\]

When the RTI interrupt function returns control to the C routine, the last task the routine must perform is preparation for the next period. A two-element pipeline of the A/D reading and a four-element error pipeline are updated. Finally the “old” duty cycle value is copied into OLDDTY.

**MC68HC11N4 MATH COPROCESSOR**

The assembly PID routine uses the MC68HC11N4 math coprocessor, which is commonly referred to as an arithmetic logic unit, or ALU. The ALU performs 32/16-bit division, 16/16 multiplication, multiply-and-accumulate operations, and 16/16-bit fractional division without CPU intervention. As Figure 2 shows, the coprocessor has one control register, one status register, and three data registers. Arrows indicate the most convenient order of writing the registers.

![Figure 2: Coprocessor Registers and Operations](image-url)
The 8-bit ALU control register (ALUC) controls ALU operation. Table 1 summarizes ALUC bit functions.

### ALUC — Arithmetic Logic Unit Control

<table>
<thead>
<tr>
<th>Bit 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>SIG</td>
<td>DIV</td>
<td>MAC</td>
<td>DCC</td>
<td>TRG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RESET:**

0 0 0 0 0 0 0 0

**SIG** — Signed Number Enable
- **0** = AREG, BREG, and CREG contents are unsigned numbers
- **1** = AREG, BREG, and CREG contents are signed numbers

**DIV** — Division Enable

**MAC** — Multiply with Accumulated Product Enable

**DCC** — Division Compensation for Concatenated Quotient Enable

**TRG** — Function Start Trigger Bit
- **Always reads zero**
- **0** = No effect
- **1** = Start ALU

**Bits [2:0]** — Not implemented
- **Always read zero**

### Table 1 ALUC Bit Function

<table>
<thead>
<tr>
<th>SIG</th>
<th>DIV</th>
<th>MAC</th>
<th>DCC</th>
<th>FUNCTION</th>
<th>START TRIGGERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Unsigned MUL</td>
<td>Write BREG or set TRG</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Signed MUL</td>
<td>Write BREG or set TRG</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>Unsigned MAC</td>
<td>Write BREG or set TRG</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>Signed MAC</td>
<td>Write BREG or set TRG</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Unsigned IDIV</td>
<td>Write AREG or set TRG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Signed IDIV</td>
<td>Write AREG or set TRG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Signed IDIV DCC</td>
<td>Write AREG or set TRG</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Unsigned FDIV</td>
<td>Set TRG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Signed FDIV</td>
<td>Set TRG</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Signed FDIV DCC</td>
<td>Set TRG</td>
</tr>
</tbody>
</table>

The 8-bit ALU status register (ALUF) signals when an operation is complete and indicates the status of the completed operation.

### ALUF — Arithmetic Logic Unit Status Flag Register

<table>
<thead>
<tr>
<th>Bit 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>NEG</td>
<td>RZF</td>
<td></td>
<td></td>
<td></td>
<td>OVF</td>
<td>DZF</td>
<td>ACF</td>
</tr>
</tbody>
</table>

**RESET:**

0 0 0 0 0 0 0 0

**NEG** — Negative Result
- NEG is set if the result is a negative value. This is a read-only-bit. Writes to this bit do not affect the value.

**RZF** — Remainder Equals Zero Flag
- RZF is set if the remainder is zero.

**Bits [5:3]** — Not implemented
- **Always read zero**
OVF — Overflow Flag
OVF is set if overflow from MSB on CREG is detected. This bit is cleared automatically by a write to this register with bit 2 set.

DZF — Divide by Zero Flag
DZF is set if a divide by zero condition is detected. DZF is cleared automatically by a write to this register with bit 1 set.

ACF — Arithmetic Completion Flag
ACF is set by completion of the arithmetic operation. ACF is cleared automatically by a write to this register with bit 0 set.

Data register A (AREG) can hold either a 16-bit multiplicand or a 16-bit divisor. Data register B (BREG) can hold either a 16-bit multiplier or a 16-bit remainder after division. Data register C (CREG), which is treated as two 16-bit registers (CREG High and CREG Low), can hold a 32-bit product or accumulated product after multiplication, or it can hold a 32-bit numerator before division and a 32-bit quotient after division. There is an implied fixed radix point to the right of bits AREG0, BREG0, and CREG0.

Table 2 shows numeric ranges of ALU registers. Table 3 shows signed expression of numbers. Table 4 shows fractional numeric representation. Figure 3 shows typical data register formats.

### Table 2 Numeric Ranges of ALU Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Size</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREG</td>
<td>16 bits</td>
<td>0 to 65535</td>
<td>–32,768 to +32,767</td>
</tr>
<tr>
<td>BREG</td>
<td>16 bits</td>
<td>0 to 65535</td>
<td>–32,768 to +32,767</td>
</tr>
<tr>
<td>CREG</td>
<td>32 bits</td>
<td>0 to 4,294,967,295</td>
<td>–2,147,483,648 to +2,147,483,647</td>
</tr>
</tbody>
</table>

### Table 3 Representation of Signed Numbers

<table>
<thead>
<tr>
<th>Decimal</th>
<th>16-Bit Hexadecimal</th>
<th>32-Bit Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2,147,483,647</td>
<td>—</td>
<td>$7FFF FFFF</td>
</tr>
<tr>
<td>·</td>
<td>—</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>—</td>
<td>·</td>
</tr>
<tr>
<td>+32,767</td>
<td>$7FF</td>
<td>$0000 7FF</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>+1</td>
<td>$0001</td>
<td>$0000 0001</td>
</tr>
<tr>
<td>0</td>
<td>$0000</td>
<td>$0000 0000</td>
</tr>
<tr>
<td>–1</td>
<td>$FFFF</td>
<td>$FFFF FFFF</td>
</tr>
<tr>
<td>–2</td>
<td>$FFFE</td>
<td>$FFFF FFFE</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>–32,768</td>
<td>$8000</td>
<td>$FFFF 8000</td>
</tr>
<tr>
<td>·</td>
<td>—</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>—</td>
<td>·</td>
</tr>
<tr>
<td>–2,147,483,647</td>
<td>—</td>
<td>$8000 0000</td>
</tr>
</tbody>
</table>
Table 4 Representation of Fractions

<table>
<thead>
<tr>
<th>Decimal</th>
<th>16-Bit Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.99998</td>
<td>$FFFF</td>
</tr>
<tr>
<td>+0.5</td>
<td>$8000</td>
</tr>
<tr>
<td>+0.25</td>
<td>$4000</td>
</tr>
<tr>
<td>+0.125</td>
<td>$2000</td>
</tr>
<tr>
<td>+0.0625</td>
<td>$1000</td>
</tr>
<tr>
<td>+0.03125</td>
<td>$0800</td>
</tr>
<tr>
<td>+0.015625</td>
<td>$0400</td>
</tr>
<tr>
<td>-0.99998</td>
<td>$0001</td>
</tr>
<tr>
<td>-0.5</td>
<td>$8000</td>
</tr>
<tr>
<td>-0.25</td>
<td>$C000</td>
</tr>
<tr>
<td>-0.125</td>
<td>$E000</td>
</tr>
<tr>
<td>-0.0625</td>
<td>$F000</td>
</tr>
<tr>
<td>-0.03125</td>
<td>$F800</td>
</tr>
<tr>
<td>-0.015625</td>
<td>$FC00</td>
</tr>
<tr>
<td>-0.0000153</td>
<td>$FFFF</td>
</tr>
</tbody>
</table>
ASSEMBLY LANGUAGE IMPLEMENTATION

The assembly language version of the PID control routine consists of a C master program and an assembly subroutine (DOPID) that carries out all time-critical tasks (see Appendix B for a complete listing). The C master program is used for convenience — DOPID can be made to stand alone with minor changes in variable definition.

The constant and variable number representation for the assembly version of the PID loop is a four-byte number consisting of a 16-bit integer, an implied decimal point, and a 16-bit fraction. This representation was chosen to provide relatively high performance while giving enough precision to support MC68HC11N4 8-, 12-, and 16-bit PWM resolutions. Many other formats could be used, each with a particular trade-off between precision and time.
Although the sampling period constant and the constants of each term of the PID algorithm could be calculated during the assembly phase and set in the final code, these values are instead initialized as ratios of hexadecimal integers, then calculated in real time in the loop. This arrangement allows easy experimentation with these values without reassembling each time a value is changed. The initial format of these values (numerator and denominator) is a consequence of the four-byte numeric representation.

Here are some examples of ALU operations performed during PID computation.

To perform a signed integer divide followed by a signed fractional divide, the numerator is written to CREG, $C0 is written to ALUC, and the divisor is written to AREG. After the completion flag in ALUF is set ($49), $E8 is written to ALUC. Following integer division, the quotient is in CREG and the remainder is in BREG. Fractional division moves the content of CREG Low to CREG High, then places a 16-bit fraction in CREG Low. The final result is a 16-bit quotient in CREG High and a 16-bit fraction in CREG Low. There is an implicit decimal point between CREG High and CREG Low. More precision can be obtained by concatenating fractional divisions.

To perform a signed multiply, $80 is written to ALUC and multiplicands are written to AREG and BREG. When the operation is complete, a 32-bit result is in CREG.

The actual range used for control is hexadecimal $0000.0000 to $00FF.FFFF (decimal 0 to 255.99998). While the error term can be positive or negative, PWM output and feedback voltage are always positive. A result greater than $FF is treated as an overflow, and a result less than $00 is treated as an underflow. This effects a saturated value of the correct sign as in the C version. Except for the expression of initial constants as ratios, formula 5 is not changed. In the derivative term the factor

$$\frac{(K_{D\text{NUM}} / K_{D\text{DEN}})}{(6 \times \left(\frac{PERD_{\text{NUM}}}{PERD_{\text{DEN}}}\right))}$$

is rearranged to

$$\left(\frac{(K_{D\text{NUM}} / K_{D\text{DEN}}) \times PERDT_{\text{DEN}}}{(6 \times PERDT_{\text{NUM}})}\right).$$

The DOPID routine is written as straight-line code. Only two subroutines and a limit-checking section are shared by the proportional, integral, and derivative terms. The subroutines MULLNG and ADLNG are used to multiply and add terms and factors expressed in the special four-byte format explained previously. Each of the P, I, and D terms use ADLNG to contribute to the new PWM duty cycle (NEWDTY), but, as in the C version of the PID routine, NEWDTY is not output until the beginning of the next period. Only the 8-bit integer portion is used, and the 1-bit round-off error this causes is not corrected — the effect is negligible in this 8-bit example. After all three terms are calculated, control is returned to the master C routine. The master routine updates the error and A to D pipelines, then enters the main wait loop.

During program execution, all results and most intermediate values are kept in RAM, rather than on the stack. Controller state can easily be inspected by means of a single breakpoint and a dump of the appropriate variable address. Variable addresses are provided in the C startup code for the assembly routine — the addresses are only valid for this compilation and can change with code revision.

**HARDWARE PLATFORM**

The Motorola M68HC11KMNPEVS Evaluation System can be used to run both versions of the PID routine. With an MC68HC11K4 inserted in the emulator module, only the floating point version will execute. With an MC68HC11N4 inserted, both versions can be executed — the floating point version simply does not use the math coprocessor.

Both versions of the PID routine utilize special test mode in the EVS system. This means that the M68HC11 processor vectors are mapped from $BFD6 to $BFFFF instead of from $FFD6 to $FFFF, and can be placed in user RAM or in emulation RAM, making experimentation with varied processor configuration options easier. Refer to the *M68HC11KMNPEVS Evaluation System User's Manual* for more information.
S-record formatted object code can be loaded and run on the EVS using appropriate serial communications software. The default EVS Baud rate is 19200 Baud.

A simple RC circuit is used to provide the feedback necessary for the software PID loop to operate. The connections to the EVS are shown in Figure 4. PH0 is the processor PWM output and PE0 is the channel 1 A/D input. Note that the A/D reference inputs must be connected to appropriate supplies and the SUP6(IRQ) line must also be tied high. A two-channel oscilloscope can be used to observe PWM output and controlled voltage. Both versions of the code write $FF to PORTA just before performing PID loop calculations and then write $00 to the port just after calculations are complete, allowing execution times to be observed with a scope.

![Evaluation System Schematic](image)

**Figure 4 Evaluation System Schematic**

**PERFORMANCE**

Since two very different approaches were taken to implement the PID loop algorithm, it is not surprising that the performance and code sizes of the two routines differ significantly. The C language version contains over 1500 bytes, much of which consists of floating point runtime support. Because it carries full floating-point precision and does not use math support hardware, the C version takes approximately 6 milliseconds to complete the loop. The assembly version contains approximately 1000 bytes, and could be reduced to about 800 bytes if features added for clarity and experimentation were removed. Because it uses a tailored arithmetic format and gets a hardware assist from the math coprocessor, the assembly version completes the loop in approximately 700 microseconds. In both cases, performance could be improved by precomputing all the results with constant factors.

**EXPERIMENTS AND EXTENSIONS**

To develop a more intuitive understanding of PID loop function, try varying the term constants in the object code and observing the effect on controller performance. For the C version, it is best to change the values in the C source and recompile. For the assembly version, it is easy to recalculate and change denominators of the constants: only decimal-to-hex conversion is required. The PERDT constant must be changed when the RTI interrupt rate is changed, or the time base of the algorithm will be destroyed. Try adjusting the proportional and derivative constants to give good observable control and then start substituting smaller value resistors for R1. Eventually, the substitution will cause an unstable underdamped system.
Many extensions to the routines are simple yet interesting. The command voltage level can be read from one of the seven unused A/D channels to give real-time control points. The MC68HC11N4 version could have analog output instead of PWM with the use of an 8-bit D/A channel. The N4 version could also use an infinite impulse response (IIR) filter implemented by means of the math coprocessor multiply-and-accumulate capability, rather than utilizing the noise-canceling effect of the derivative term's four point central difference.

**CONCLUSION**

With the 16.383 millisecond sampling rate chosen, the floating point C routine cannot service all four MC68HC11K4 PWM channels. It could, however, service one higher-rate channel and several less time-critical processes. The assembly math coprocessor routine can service all the MC68HC11N4 PWM channels (four 8-bit and two 12-bit) in 4.2 milliseconds, leaving 75% of execution time for other tasks. If the MC68HC11N4 D/A channels were also used, six independent PID controllers could be run in 5.6 milliseconds — this routine would use less than a third of the allowable processing time and require only half of the twelve MC68HC11N4 A/D input channels.

The two approaches to PID control outlined in this note encompass a wide range of useful and cost-effective applications. The simplicity of the coded C algorithm is very appealing. Since the C routine carries virtually no code-space overhead, it is very well-suited to an application where floating-point math is already required, and a sampling rate in the 20-millisecond range is acceptable. Because it uses a number format adapted to the application and to use of on-chip resources, the assembly routine generally yields higher, more cost-effective performance than the C routine.

A final word of caution. There is no substitute for thorough mathematical analysis of the system to be controlled in the discrete time domain. References 2 and 3 contain detailed discussions of control systems and analytic techniques.

**REFERENCES**

4. Motorola. MC68HC11K4 Technical Summary, BR751/D
5. Motorola. MC68HC11N4 Technical Summary, MC68HC11N4TS/D
APPENDIX A
C LANGUAGE PID ROUTINE

A.1 Main C Routine

```c
#include <stdio.h>
#include <io6811k4.h>
#include <int6811k.h>
#include <math.h>

unsigned int TOFCOUNT; /* declare variables */
float CMNDVX;
float ADRCX;
float ADRCXM1;
float ADRCXM2;
float ADRCXM3;
float ERRX;
float ERRM1X;
float ERRM2X;
float ERRM3X;
float PERDT;
float NEWDTY;
float OLDDTY;
float KP;
float KD;
float KI;

extern int DOIO (void); /* prototype for assembly routine */

void main() /* main program */
{
    CMNDVX = 1.5;
    PERDT = 0.016383; /* RTI and therefore PID loop period = 16.383 ms */
    KP = 0.18; /* kp = .12, ki = 6.0, kd = .006, for 1 M ohm drive */
    KI = 6.0;
    KD = 0.009;
    OLDDTY = 1.9; /* start out with pwm set fairly high */
    PORTA = 0x00; /* this will be used for a scope trigger */
    DDRA = 0xFF; /* set PORTA as output */
    PACTL = 0x03; /* set RTI to 16.383 ms (E = 4 MHz) */
    TMSK2 = 0x40; /* enable RTI interrupts */
    OPTION = 0x90; /* enable A/D charge pump */
    PWPER1 = 0xFF; /* set up PWM channel 1 at 15.625 kHz */
    PWDTY1 = 0xFF; /* with positive polarity */
    PWPOL = 0x01;
    DDRH = 0x00;
    PWEN = 0x01;
    TFLG2 = 0x40;
    enable_interrupt(); /* wait here for RTI to cause loop execution */
    wait_for_interrupt();
    for (;;) {
        ;
    }
}

interrupt void IRQ_interrupt(void) /* should initialize all interrupts... */
{
    PORTA = 0xFF;
    PORTA = 0x00;
}

interrupt void TO_interrupt(void)
{
    TOFCOUNT++ ;
}
```
interrupt void RTI_interrupt(void) /*PID LOOP/PWM routine */
{
    PORTA = 0xFF; /* scope strobe */
    DOIO(); /* read A to D and output the duty cycle calculated last period */
    ADCTL = 0x10; /* begin new conversion cycle */
    ERRX = (CMNDVX - ADRCX); /* calculate current error */
    /* The statement below is the entire floating point PID loop */
    NEWDTY = KP*(ERRX) + KI*PERDT*(CMNDVX - (ADRCX + ADRCXM1)/2) + (KD/(6*PERDT))*((ERRX - ERRM3X) + 3*(ERRM1X - ERRM2X)) + OLDDTY;
    if (NEWDTY > 1.99609) /* test for result being in usable */
        NEWDTY = 1.99609; /* limits and set PWM duty cycle if */
    else if (NEWDTY < 1.0) /* beyond saturation */
        NEWDTY = 1.0;
    TFLG2 = 0x40; /* clear RTI flag */
    ADRCXM1 = ADRCX; /* update A/D result for next cycle */
    ERRM3X = ERRM2X; /* update error pipeline */
    ERRM2X = ERRM1X;
    ERRM1X = ERRX;
    OLDDTY = NEWDTY; /* update old duty cycle for next calculation period */
    PORTA = 0x00; /* scope strobe */
}

A.2 DOIO Subroutine Assembler Listing

1  ******************************************************************************
2  * DOIO assembly function  *
3  * This routine handles the conversion between  *
4  * 8 bit register values and the C float variables  *
5  *  *
6  ******************************************************************************
7
8 0000      MODULE  DOIO
9 0000      PUBLIC  DOIO
10
11 0000      P68H11
12 0000      RSEG    CODE
13
14 006C      PWDTY1  set     $006c      REGISTER LOCATIONS
15 0031      ADDR1   set     $0031
16 0000      EXTERN  ADRCX:ZPAGE  EXTERNAL VARIABLE LOCATIONS
17 0000      EXTERN  NEWDTY:ZPAGE
18 0000      DOIO:
19 0000 863F    LDAA    #$3F      INITIALIZE FLOAT LOCATION.
20 0002 5F     CLRb
21 0003 D000    STD    ADRCX      GET CHANNEL 1 A/D RESULT.
22 0005 9631    LDAA   ADR1
23 0007 04    LSRD      SHIFT TO FLOAT MANTISSA POSITION.
24 0008 8A80    CRAA   #$80      OR IN LEAST SIGNIFICANT EXP BIT
25 000A D001    STD    ADRCX+1    AND STORE IT IN FLOAT VARIABLE.
26 000C 5F     CLRb      CLEAR LEAST SIGNIFICANT
27 000D D703    STAB   ADRCX+3    FLOAT BYTE.
28 000F DC01    LDD    NEWDTY+1    GET TWO BYTES OF FLOAT MANTISSA.
29 0011 05    LSLD      SHIFT TO CORRECT REGISTER POSITION.
30 0012 976C    STAA   PWDTY1    OUTPUT TO PWM DUTY REGISTER.
31 0014 39    RTS
32 0015      END
APPENDIX B
ASSEMBLY LANGUAGE PID ROUTINE

B.1 Master C Routine

/* This code just sets up variables and does some updates
after the assembly PID loop is called. */

#include <stdio.h>
#include <io6811N4.h>
#include <int6811N.h>

unsigned int TOFCOUNT;
signed int CMNDVX;
signed int ADRCX;
signed int ADRCXM1;
signed int ERRX;
signed int ERRM1X;
signed int ERRM2X;
signed int ERRM3X;
signed int KPNUM;
signed int KDNUM;
signed int KINUM;
signed int KIDEN;
signed int KDNUM;
signed int PERDTNUM;
signed int PERDTDEN;
signed int INT56;
signed int FC56;
signed int TEMP1;
signed int TEMP2;
signed int TEMP3;
signed int TEMP4;
long NEWDTY;
long OLDDTY;
long KPTRM;
long KDTRM;
long KITRM;
long LTEMP1;
long LTEMP2;
long LTEMP3;
long LTEMP4;
long LTEMP5;
long LTEMP6;
long LTEMP7;
long LTEMP8;
long LTEMP9;
long LTEMPA;
long FCINT56;
long INTFC56;

extern int DOPID (void); /* prototype for assembly routine */
void main() /* main program */
{
  CMNDVX = 0x0080;
  PERDTNUM = 0x00A4;
  PERDTDEN = 0x2710;  /* PERIOD = 164/10000 decimal */
  KPNUM = 0x000C;
  KPDEN = 0x0064;   /* kp = .12, ki = 6.0, kd = .006, for 1M ohm drive */
  KINUM = 0x0006;
  KIDEN = 0x0001;
  KDNUM = 0x0006;
  KDDEN = 0x003E8;
  OLDDTY = 0x00FF0000;
  PORTA = 0x00;
  DDRA = 0xFF;     /* set PORTA as output */
  PACTL = 0x03;    /* set RTI to 16.383 ms (E = 4MHz) */
  TMSK2 = 0x40;    /* enable RTI interrupts */
  OPTION = 0x90;   /* enable A/D charge pump */
  PWPER1 = 0xFF;   /* set up PWM channel 1 at 15.625 kHz */
  PWDTY1 = 0xFF;   /* with positive polarity */
  PWPOL = 0x01;
  DDRH = 0x00;
  PWEN = 0x01;
  TFLG2 = 0x40;

  enable_interrupt(); /* wait for RTI to execute assembly PID routine */

  wait_for_interrupt();

  for (;;) {
    
    interrupt void IRQ_interrupt(void) /* Just some traps for unexpected */
    {
      PORTA = 0xFF;
      PORTA = 0x00;
    }

    interrupt void TO_interrupt(void)
    {
      TOFCOUNT++;
    }

    interrupt void RTI_interrupt(void) /* PWM routine */
    {
      PORTA = 0xFF;
      DOPID(); /* DO THE PID LOOP USING THE MATH COPROCESSOR */
      NEWDTY = KP*(ERRX) + KI*PERDT*(CMNDVX - (ADRCX + ADRCXM1)/2)
        + (KD/(6*PERDT))*((ERRX - ERRM3X) + 3*(ERRM1X - ERRM2X))
        + OLDDTY; */
      TFLG2 = 0x40; /* clear RTI flag */
      ADRCXM1 = ADRCX; /* update A/D result for next cycle */
      ERRM3X = ERRM2X; /* update error pipeline */
      ERRM2X = ERRM1X;
      ERRM1X = ERRX;
      PORTA = 0x00; /* scope strobe */
  }
}
B.2 DOPID Assembly Listing

****************************************************************

* DOPID assembly function*
* These routines calculate the new PWM duty cycle*
* using the MC68HC11N4 math coprocessor. The*
* code can be run on an M68HC11EVS with an*
* M68HC11K4 emulator and MC68HC11N4 processor.*
* The EVS monitor should be 1.1 or later. The EVS*
* and vectors were set to SPECIAL TEST MODE to aid debug.*
* This code is called by a C routine but could be converted*
* to an all assembly environment by defining the variables*
* in assembly instead of as externals.*
****************************************************************

MODULE  DOPID
PUBLIC  DOPID

P68H11
RSEG  CODE

EXTERN  ADRCX:ZPAGE            $0084  EXTERNAL VARIABLES
EXTERN  ADRCXM1:ZPAGE          $0086  SIGNED INTS
EXTERN  CMNDVX:ZPAGE           $0082
EXTERN  ERRX:ZPAGE             $0088
EXTERN  ERRM1X:ZPAGE           $008A
EXTERN  ERRM2X:ZPAGE           $008C
EXTERN  ERRM3X:ZPAGE           $008E
EXTERN  KNUM:ZPAGE             $0090
EXTERN  KPDEN:ZPAGE            $0092
EXTERN  KNUM:ZPAGE             $0094
EXTERN  KIDEN:ZPAGE            $0096
EXTERN  KDNUM:ZPAGE            $0098
EXTERN  KDNUM:ZPAGE            $009A
EXTERN  PERDTNUM:ZPAGE         $009C
EXTERN  PERDTDEN:ZPAGE         $009E
EXTERN  INT56:ZPAGE            $00A0
EXTERN  FC56:ZPAGE             $00A2
EXTERN  TEMP1:ZPAGE            $00A4
EXTERN  TEMP2:ZPAGE            $00A6
EXTERN  TEMP3:ZPAGE            $00A8
EXTERN  TEMP4:ZPAGE            $00AA
EXTERN  KPTRM:ZPAGE            $00B4  LONGS
EXTERN  KITRM:ZPAGE            $00BC
EXTERN  KDTRM:ZPAGE            $00B8
EXTERN  LTEMP1:ZPAGE           $00C0
EXTERN  LTEMP2:ZPAGE           $00C4
EXTERN  LTEMP3:ZPAGE           $00C8
EXTERN  LTEMP4:ZPAGE           $00CC
EXTERN  LTEMP5:ZPAGE           $00D0
EXTERN  LTEMP6:ZPAGE           $00D4
**DOPID:**

******** OUTPUT LAST PERIOD RESULT AND DO KP TERM ********

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>0000 9601</td>
</tr>
<tr>
<td>79</td>
<td>0002 976C</td>
</tr>
<tr>
<td>80</td>
<td>0004 4F</td>
</tr>
<tr>
<td>81</td>
<td>0005 D631</td>
</tr>
<tr>
<td>82</td>
<td>0007 DD00</td>
</tr>
<tr>
<td>83</td>
<td>0009 C610</td>
</tr>
<tr>
<td>84</td>
<td>000B D730</td>
</tr>
<tr>
<td>85</td>
<td>000C DC00</td>
</tr>
<tr>
<td>86</td>
<td>000F 9300</td>
</tr>
<tr>
<td>87</td>
<td>0011 DD00</td>
</tr>
<tr>
<td>88</td>
<td>0013 2B06</td>
</tr>
<tr>
<td>89</td>
<td>0015 8600</td>
</tr>
<tr>
<td>90</td>
<td>0017 9700</td>
</tr>
<tr>
<td>91</td>
<td>0019 2004</td>
</tr>
<tr>
<td>92</td>
<td>001B 86FF</td>
</tr>
<tr>
<td>93</td>
<td>001D 9700</td>
</tr>
<tr>
<td>94</td>
<td>001F 8680</td>
</tr>
<tr>
<td>95</td>
<td>0021 9744</td>
</tr>
<tr>
<td>96</td>
<td>0023 DC00</td>
</tr>
<tr>
<td>97</td>
<td>0025 DD45</td>
</tr>
<tr>
<td>98</td>
<td>0027 8601</td>
</tr>
<tr>
<td>99</td>
<td>0029 9749</td>
</tr>
<tr>
<td>100</td>
<td>002B DC00</td>
</tr>
<tr>
<td>101</td>
<td>002D DD47</td>
</tr>
<tr>
<td>102</td>
<td>002F 134901FC WPMUL1 BRC LR ALUF,#$01,WPMUL1 WAIT FOR ACF</td>
</tr>
<tr>
<td>103</td>
<td>0033 86C0</td>
</tr>
<tr>
<td>104</td>
<td>0035 DD44</td>
</tr>
<tr>
<td>105</td>
<td>0037 8601</td>
</tr>
<tr>
<td>106</td>
<td>0039 9749</td>
</tr>
<tr>
<td>107</td>
<td>003B DC00</td>
</tr>
<tr>
<td>108</td>
<td>003D DD45</td>
</tr>
<tr>
<td>109</td>
<td>003F 134901FC WPDIV1 BRC LR ALUF,#$01,WPDIV1 WAIT FOR ACF</td>
</tr>
<tr>
<td>110</td>
<td>0043 8601</td>
</tr>
<tr>
<td>111</td>
<td>0045 9749</td>
</tr>
<tr>
<td>112</td>
<td>0047 86B8</td>
</tr>
<tr>
<td>113</td>
<td>0049 DD44</td>
</tr>
<tr>
<td>114</td>
<td>004B 134901FC WPFDV1 BRC LR ALUF,#$01,WPFDV1 WAIT FOR ACF</td>
</tr>
<tr>
<td>115</td>
<td>004F DC40</td>
</tr>
<tr>
<td>116</td>
<td>0051 DD00</td>
</tr>
<tr>
<td>117</td>
<td>0053 DD00</td>
</tr>
<tr>
<td>118</td>
<td>0055 DC42</td>
</tr>
<tr>
<td>119</td>
<td>0057 DD02</td>
</tr>
<tr>
<td>120</td>
<td>0059 DD02</td>
</tr>
<tr>
<td>121</td>
<td>005B BD0361</td>
</tr>
<tr>
<td>122</td>
<td>005E BD007F</td>
</tr>
<tr>
<td>123</td>
<td>0061 BD014F</td>
</tr>
<tr>
<td>124</td>
<td>0064 DC00</td>
</tr>
<tr>
<td>125</td>
<td>0066 2B0F</td>
</tr>
<tr>
<td>126</td>
<td>0068 1A8300FF</td>
</tr>
<tr>
<td>127</td>
<td>006C 2B10</td>
</tr>
<tr>
<td>128</td>
<td>006E CC00FF</td>
</tr>
<tr>
<td>129</td>
<td>0071 DD00</td>
</tr>
<tr>
<td>130</td>
<td>0073 DD00</td>
</tr>
<tr>
<td>131</td>
<td>0075 2007</td>
</tr>
<tr>
<td>132</td>
<td>0077 CC0000 JAM2P LDD #$0000 JAM 00</td>
</tr>
<tr>
<td>133</td>
<td>007A DD00</td>
</tr>
<tr>
<td>134</td>
<td>007C DD00</td>
</tr>
</tbody>
</table>
* ROUTINE TO DO INTEGRAL TERM *

GET CURRENT CONVERSION

FORM (ADRCX + ADRCXM1)/2

TERM WILL ALWAYS BE 0 or 0.5

CMNDVX - ((ADRCX + ADRCXM1)/2)
203 010F DD00 STD LTEMP5
204 0111 DC02 LDD LTEMP3+2
205 0113 DD00 STD LTEMP5+2
206 0115 DC00 LDD LTEMP4
207 0117 DD00 STD LTEMP6
208 0119 DC02 LDD LTEMP4+2
209 011B DD02 STD LTEMP6+2
210 011D 9600 LDAA TEMP3 SAVE SIGN FLAG
211 011F 9700 STAA TEMP4 AND USE TEMP3 AS A FLAG
212 0121 8600 LDAA #$00 FOR LTEMP6 BEING POSITIVE
213 0123 9700 STAA TEMP3
214 0125 BD023A JSR MULLNG DO LTEMP3*LTEMP4(PERDT*KI)
215 0128 DC00 LDD LTEMP7 NOW PUT RESULT IN LTEMP5
216 012A DD00 STD LTEMP6
217 012C DC02 LDD LTEMP7+2
218 012E DD02 STD LTEMP6+2
219 0130 9600 LDAA TEMP4 RETRIEVE SIGN FLAG
220 0132 9700 STAA TEMP3
221 0134 DC00 LDD LTEMP2
222 0136 DD00 STD LTEMP6 ERROR FOR KI TERM
223 0138 DC02 LDD LTEMP2+2
224 013A DD02 STD LTEMP6+2
225 013C BD023A JSR MULLNG DO RESULT*LTEMP2
226 013F DC00 LDD LTEMP7
227 0141 DD00 STD LTEMP1
228 0143 DD00 STD KITRM
229 0145 DC02 LDD LTEMP2+2
230 0147 DD02 STD LTEMP6+2
231 0149 DD02 STD KITRM+2 ADD KI TERM INTO NEWDTY
232 014B BD0361 JSR ADLNG KITERM DONE
233 014E 39 RTS RETURN

********** ROUTINE TO DO KD TERM **********

236 014F DC00 DOKDT LDD ERRX FORM (ERRX - ERRM3X)
237 0151 9300 SUBD ERRM3X + 3*(ERRM1X - ERRM2X)
238 0153 DD00 STD TEMP1
239 0155 DC00 LDD ERRM1X
240 0157 9300 SUBD ERRM2X
241 0159 DD45 STD AREGH FORM 3*(ERRM1X - ERRM2X)
242 015B 8680 LDAA #$80
243 015D 9744 STAA ALUC
244 015F 8601 LDAA #$01
245 0161 9749 STAA ALUF
246 0163 CC0003 LDD #$0003
247 0165 DD47 STD BREGH
248 0168 134901FC WDMUL0 BRCLR ALUF, #$01, WDMUL0 WAIT FOR ACF
249 016C DC42 LDD CREGML
250 016E D300 ADDD TEMP1
251 0170 DD00 STD LTEMPA
252 0172 2B02 BMI NGFLGS0 SET UP SIGN FLAG IN TEMP3
253 0174 2006 BRA POSGN
254 0176 86FF NGFLGS0 LDAA #$FF
255 0178 9700 STAA TEMP3
256 017A 2004 BRA KDFLGD
257 017C 8600 POSGN LDAA #$00
258 017E 9700 STAA TEMP3
259 0180 CC0000 KDFLGD LDD #$0000 DONE
260 0182 DD02 STD LTEMPA+2
261 0184 8600 LDAA #$00 FORM 6*PERDTNUM
262 0186 9744 STAA ALUC
263 0188 8601 LDAA #$01
264 018A 9749 STAA ALUF
265 018B 9749 STAA ALUF
266 018D CC0006 LD #0006
267 0190 DD45 STD AREGH
268 0192 DC00 LDD PERDTNUM
269 0194 DD47 STD BREGH
270 0196 134901FC WDMUL1 BRCLR ALUF, #$01, WDMUL1 WAIT FOR ACF
339 022E DD00  STD  KDTRM
340 0230 DC02  LDD  LTEMP7+2
341 0232 DD02  STD  LTEMP1+2
342 0234 DD02  STD  KDTRM+2
343 0236 BD0361  JSR  ADLNG  ADD KD TERM INTO NEWDTY
344 0239 39  RTS  KDTERM DONE

345

346  * SUBROUTINE TO MULTIPLY LONGS (INTEGER & FRACTION) *
347  * LTEMP5*LTEMP6=LTEMP7 ONLY LTEMP6 CAN HAVE *
348  * A NEGATIVE TERM TO HANDLE. *
349

350 023A 8680  MULLNG  LDAA  #$80  SET ALU FOR SMUL
351 023C 9744  STAA  ALUC  AND MULTIPLY INTS
352 0240 DD45  STD  AREGH
353 0242 8601  LDAA  #$01  CLEAR ACF
354 0244 9749  STAA  ALUF
355 0246 DC00  LDD  LTEMP6
356 0248 DD47  STD  BREGH  TRIGGER SMUL
357 0250 134901FC  WMULL1  BRCLR  ALUF,#$01,WMULL1  WAIT FOR ACF
358 024E DC42  LDD  CREGML
359 0250 DD00  STD  INT56
360 0252 8601  LDAA  #$01  CLEAR ACF AND DO NEXT MULT
361 0254 9749  STAA  ALUF
362 0256 8680  LDAA  #$80  TEST TEMP3 SIGN
363 0258 9500  BITA  TEMP3  SEE IF ERR IS NEG
364 025A 2B12  BMI  NEGFRACT  TERM IS NEGATIVE
365 025C 8601  LDAA  #$01  CLEAR ACF
366 025E DC02  LDD  LTEMP6+2  GET FRAC NOT NEG
367 0260 8680  LDAA  #$80  TEST TEMP3 SIGN
368 0262 9500  BITA  TEMP3  SEE IF SIGN OVERFLOW
369 0264 2B0F  BMI  DFXINT  ON FRACTION
370 0266 8680  LDAA  #$80  SEE IF SIGN OVERFLOW
371 0268 9545  BITA  AREGH  ON FRACTION
372 026A 2B02  BMI  FXINT  ON FRACTION
373 026C 2007  BRA  DFXINT
374 026E CC0000  NEGFRACT  LDD  #$0000  NEGATE FRAC
375 0271 9302  SUBD  LTEMP6+2
376 0273 DD47  STD  BREGH  TRIGGER SMUL
377 0275 134901FC  WMULL3  BRCLR  ALUF,#$01,WMULL3  WAIT FOR ACF
378 0279 CC0000  LDD  #$0000  NEGATE RESULT
379 027C 9340  SUBD  CREGML  SCALE AND STORE
380 027E 2B02  BMI  INTFIX1
381 0280 2003  BRA  INTFIX2
382 0282 C30001  INTFIX1 ADDD  #$0001
383 0285 DD00  STD  INTFC56
384 0287 CC0000  LDD  #$0000
385 028A 9342  SUBD  CREGML
386 028C DD02  STD  INTFC56+2
387 028E 8600  LDAA  #$00
388 0290 DC02  NXFRAC  LDD  LTEMP5+2
389 0292 DD45  STD  AREGH  WITH POSSIBLE NEG INT
390 0294 8601  LDAA  #$01  CLEAR ACF
391 0296 9749  STAA  ALUF
392 0298 8600  LDAA  #$00
393 029A DC00  LDD  LTEMP6
394 029C DD47  STD  BREGH  TRIGGER SMUL
395 029E 134901FC  WMULL4  BRCLR  ALUF,#$01,WMULL4  WAIT FOR ACF
396 02A2 8680  LDAA  #$80
397 02A4 9500  BITA  TEMP3
398 02A6 2A0F  BPL  DFXINT
399 02A8 8680  LDAA  #$80  SEE IF SIGN OVERFLOW
400 02AA 9545  BITA  AREGH  ON FRACTION
401 02AC 2B02  BMI  FXINT
402 02AE 2007  BRA  DFXINT
403 02B0 CC0000  FXINT  LDD  #$0000
404 02B3 9340  SUBD  CREGML
405 02B5 DD40  STD  CREGML
406 02B7 DC40  DFXINT  LDD  CREGML  THIS SCALES FRAC
MOTOROLA AN1215/D

NOW DO FRAC*FRAC

CLEAR ACF

SET UNSIGNED MULT FOR FRACS

WAIT FOR ACF

SCALE AND STORE

NOW SUMM ALL PRODUCTS

INTS ARE ALL SIGNED

CAN JUST ADD UP

NOW SUMM MULT FOR FRACS

NEGATE FRAC

WAIT FOR ACF

NEGATE RESULT

NOW SUMM ALL PRODUCTS

FRACS ARE NEGATIVE

POSITIVE

NEGATE FRACS

FRACS ARE NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE

NEGATIVE
* SUBROUTINE TO ADD INTEGER AND FRACTION IN LTEMP1 TO OLDDTY *

489

490 0361 8680  ADLNG  LDAA  #$80  TEST ERRX SIGN
491 0363 9500  BITA  TEMP3
492 0365 2B19  BMI  KXNEG  TERM IS NEGATIVE
493 0367 DC00  LDD  LTEMP1  GET INT PART
494 0369 D300  ADDD  OLDDTY  ADD AND STORE INT
495 036B DD00  STD  NEWDTY
496 036D DC02  LDD  LTEMP1+2  GET FRAC PART
497 036F D302  ADDD  OLDDTY+2  ADD AND STORE FRAC
498 0371 DD02  STD  NEWDTY+2
499 0373 2502  BCS  INCINT
500 0375 201E  BRA  ADDONE
501 0377 CC0001  INCINT  LDD  #$0001  ADD CARRY FROM FRAC
502 037A D300  ADDD  NEWDTY
503 037C DD00  STD  NEWDTY
504 037E 2015  BCS  ADDONE
505 0380 DC00  KXNEG  LDD  LTEMP1  GET INT PART
506 0382 D300  ADDD  OLDDTY  ADD AND STORE INT
507 0384 DD00  STD  NEWDTY
508 0386 DC02  LDD  LTEMP1+2  GET FRAC PART
509 0388 D302  ADDD  OLDDTY+2  ADD AND STORE FRAC
510 038A DD02  STD  NEWDTY+2  ACTUALLY A SUBTRACTION
511 038C 2507  BCS  ADDONE
512 038E CFFFFF  DECINT  LDD  #$FFFF  SUBTRACT BORROW FROM FRAC
513 0391 D300  ADDD  NEWDTY
514 0393 DD00  STD  NEWDTY
515 0395 DC00  ADDONE  LDD  NEWDTY  UPDATE OLDDTY FOR NEXT TERM
516 0397 DD00  STD  OLDDTY  OR FINISH
517 0399 DC02  LDD  NEWDTY+2
518 039B DD02  STD  OLDDTY+2
519 039D 39  RTS
520 039E END

Errors:  None
Bytes:  926  # DOPID #
CRC:  EC21  #######