DSP32C Precision Sine Wave Generation

This note describes a precision sine wave generator program suitable for use with D/A converters hooked up to the DSP32C. While simple in principle, precision sine wave generation does require some careful programming.

When driving a D/A converter from the DSP32C, successive digital values must be sent to the converter describing the amplitude of the output wave at any given instant of time. The program discussed here does so by using a polynomial approximation to evaluate the sine.

For low precision applications a variety of methods for generating a sine wave may come to mind. These would include table look up, interpolation, and incremental methods based on the addition formulas or differential equations for sine and cosine. The problem with these methods is that they can result in tables of considerable size, interpolation noise, or accumulated incremental error.

A better solution is to compute a precision polynomial approximation at each point. This guarantees high accuracy, and is easily implemented on DSP32C. In C pseudo code one might write:

```
for ( x = 0 ; x < TWOPI ; x += delta )

        { value = sine_polynomial( x); }
```

While theoretically acceptable, in practice even this algorithm will be in error no matter how accurate the sine polynomial is. The simple step of incrementing x by adding on a delta increment results in considerable *accumulated* error in the angle x by the end of each cycle. The error is enough to produce unacceptable values for even 10 bit work. No matter how accurate the sine polynomial, errors in the input angle will directly translate to final output errors.

The best way to compute the angle argument is to carry along an integer counter and have the DSP32C use its floating point arithmetic to compute the input angle x on the fly. In C pseudo code:

```
for ( i = 0 ; i < N_TWOPI ; i++ )

        { x = x0 + i*delta; output_value = sine_polynomial( x); }
```

Algorithms that follow this pattern can generate sine wave values that fully exhaust the 24 bit mantissa accuracy of the DSP32C. The following program has a while loop that computes a single cycle of a sine wave that is buried inside an infinite loop cycling over and over. The trig function uses polynomial evaluation to compute cosine and sine on the interval -PI to PI. The coefficients were developed using a QR algorithm to compute an L2 fit with an accuracy of about 1.0E-6. The trig routine computes both sine and cosine at the same time using straightforward

Horners rule. By changing the coefficients and number of terms, other functions could be approximated.

```
main {

        register

                n               = r1,
                i               = r2,
                dtheta          = r3,
                one             = r4,
                zero            = r5,
                pi              = r6,
                output          = r7;


        /* Generate a continuous sine wave, cycling from -PI to PI. */

                 i = I;
            dtheta = DTHETA;

               one = ONE;
              zero = ZERO;
                pi = PI;


  trig_loop:
                 n = *N;
                *i = a0 = *zero;

           output = OUTPUT_ARRAY;   /* pointer passed to send ... */

                while ( n-- > 0 ) {

                        a0 = - *pi + a0 * *dtheta;

                        call trig;
                        call send;

                        *i = a0 = *i + *one;

                        }

        goto trig_loop; /* loop for the next sine wave cycle */

}
send {
        register output = r7;

            *output++ = a2 = a2; /* save the value of sine */

        return;
}
```

```
trig {

        /* Compute (a1,a2) = (cos( a0),sin( a0)), where -PI <= a0 <= PI. */

        register

                cc      = r12,
                sc      = r13;

        /* Evaluate (cos,sin) with polynomial approximation. */

        a3 = a0*a0;                     /* xx = a3 = theta^2 */

          cc = cos_coeff;
          sc = sin_coeff;

            a1 = *cc++ + a3 * *cc++;        /* cos = cc[4] + xx * cc[5]  */
            a2 = *sc++ + a3 * *sc++;        /* sin = sc[4] + xx * sc[5]  */
              nop;

            a1 = *cc++ + a3 * a1;           /* cos = cc[3] + xx * cos    */
            a2 = *sc++ + a3 * a2;           /* sin = sc[3] + xx * sin    */
              nop;

            a1 = *cc++ + a3 * a1;           /* cos = cc[2] + xx * cos    */
            a2 = *sc++ + a3 * a2;           /* sin = sc[2] + xx * sin    */
              nop;

            a1 = *cc++ + a3 * a1;           /* cos = cc[1] + xx * cos    */
            a2 = *sc++ + a3 * a2;           /* sin = sc[1] + xx * sin    */
              nop;

            a1 = *cc++ + a3 * a1;           /* cos = cc[0] + xx * cos    */
            a2 = *sc++ + a3 * a2;           /* sin = sc[0] + xx * sin    */
              nop;
              nop;

            a2 = a0 * a2;                   /* sin = x * sin             */

        return;  /* result is in (a1,a2) ... */

}

dspf
        cos_coeff[] = {

                -2.21543315e-007,       /* c[5] */
                 2.42466636e-005,       /* c[4] */
                -1.38624043e-003,       /* c[3] */
                 4.16609704e-002,       /* c[2] */
                -4.99995555e-001,       /* c[1] */
                 9.99999445e-001,       /* c[0] */

                },
```

```
sin_coeff[] = {

        -2.05577593e-008,      /* c[5] */
         2.70480024e-006,      /* c[4] */
        -1.98133653e-004,      /* c[3] */
         8.33259218e-003,      /* c[2] */
        -1.66665829e-001,      /* c[1] */
         9.99999732e-001,      /* c[0] */

        };
```

To turn this code into a program suitable for use with a particular D/A setup, you should modify the send routine. Typical additions would be to scale the output value and convert it to an integer before sending it off to the parallel or serial ports.