Implementation of QPSK Modem with TMS320 DSP

Georgios Vafeiadis*

Abstract

In this report, we present the implementation of $\pi/4$-DQPSK modulation and demodulation for TMS320C DSP platform of Texas Instruments. We have developed two versions. A fixed point (16-bit) optimized version and a floating point optimized version form TMS320C family of digital signal processors. The code is written in C language and you can find it in the appendix.

Keywords: Quadrature phase shift keying, QPSK Digital Transmitter/Receiver, Modulation, Demodulation, TMS320C, Texas Instruments

INTRODUCTION

In radio and optical communications the most appropriate modulation is bandpass modulation. Electromagnetic waves only propagate effectively at frequencies in the range kHz and upwards. In such situations it is necessary to modulate the baseband, message-bearing signal onto a higher frequency carrier wave. Typically, a sinusoidal carrier is used:

$$c(t) = Acos(\omega_c t + \theta) \tag{1}$$

The baseband signal occupies frequencies from zero upwards. When the baseband signal is modulated onto the carrier, its spectrum is shifted up to the carrier frequency ω_c and reflected around it. Given that a sinusoidal carrier signal is used to convey the message, there are three properties of the carrier signal that can be varied to represent the message: Amplitude, Frequency and Phase. Thus if we vary only one of these properties, there are three possible ways to modulate the message, $m(t)$ onto the carrier $c(t)$ to produce a transmitted signal $x(t)$:

Amplitude Shift Keying

$$x(t) = A(t)cos(\omega_c t + \theta) \tag{2}$$
$$A(t) = f(m(t)) \tag{3}$$

Frequency Shift Keying

$$x(t) = Acos(\omega(t)t + \theta) \tag{4}$$
$$\omega(t) = f(m(t)) \tag{5}$$

Phase Shift Keying

$$x(t) = Acos(\omega_c t + \theta(t)) \tag{6}$$
$$\theta(t) = f(m(t)) \tag{7}$$

M-ARY PSK

M-ary PSK maps b bits onto 2^b distinct phases of a carrier signal. The case of 4-ary PSK is often known as Quadrature PSK (QPSK).

Quadrature phase shift keying

Binary data bits are grouped into chunks and each chunk is mapped to a particular waveform called a symbol which is sent across the channel after modulation by the carrier. This requires having a separate symbol for each possible combination of data chunks. In QPSK, each 'chunk' consists of two bits, which determine the inphase and quadrature gain for the modulator. To produce a unique waveform for each two bit chunk, we keep the gains constant over a number of samples which is called the symbol period. To transmit two bits per symbol period, QPSK uses orthogonal waveforms (sine and cosine) referred to as the quadrature signals. The data to be transmitted on the inphase is modulated with the cosine wave, while the quadrature phase data is modulated with the sine wave. Due to the orthogonality of the sine and cosine basis functions, the data on the inphase and quadrature phase can be independently recovered at the receiver. A QPSK modulator is shown in Figure 1. QPSK modulation is achieved by essentially undertaking two BPSK modulation processes in parallel and then summing the result. A QPSK Demodulator is shown in Figure 2. Similarly to the modulation process, demodulation is based on two quadrature demodulators as employed in BSPK.

Implementation

The implementation of QPSK is more general than that of BPSK and also indicates the implementation of higher-order PSK. Writing the symbols in the constellation diagram in terms of the sine and cosine waves

*University of Bristol, MSc in Communication Systems and Signal Processing, email: gv6120@bristol.ac.uk

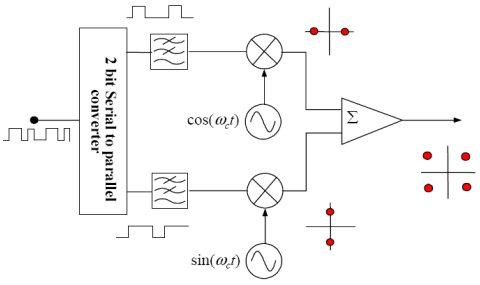

Figure 1: QPSK Modulator

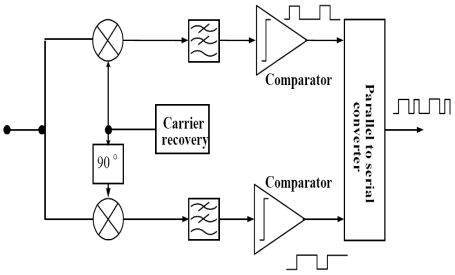

Figure 2: QPSK Demodulator

used to transmit them:

$$s_i(t) = \sqrt{\frac{2E_s}{T}} \cos\left(2\pi f_c t + (2i-1)\frac{\pi}{4}\right), i = 1, 2, 3, 4$$

This yield the four phases $\pi/4, 3\pi/4, 5\pi/4$ and $7\pi/4$.

$\pi/4$-QPSK

This variant of QPSK uses two identical constellations which are rotated by $\pi/4$ radians with respect to one another. Usually, either the even or odd data bits are used to select points from one of the constellations and the other bits select points from the other constellation. One property this modulation scheme possesses is that if the modulated signal is represented in the complex domain, it does not have any paths through the origin. In other words, the signal does not pass through the origin. This lowers the dynamical range of fluctuations in the signal which is desirable when engineering communications signals.

Results

We have implemented the modulator and demodulator in CCstudio. In order to tested, we have implemented a simple testing program. The testing program generates a random binary message, performs the encoding $\pi/4$-QPSK and the decoding. In figures 3, 5 we can see the constellation Diagram in CCstudio Graph Viewer.

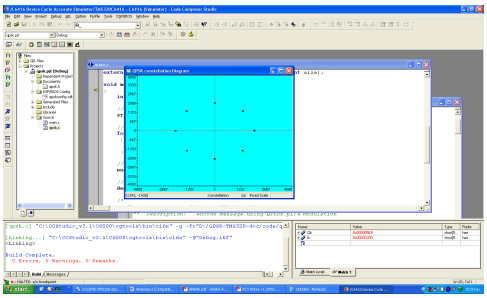

Figure 3: QPSK Constellation Diagram in CCstudio

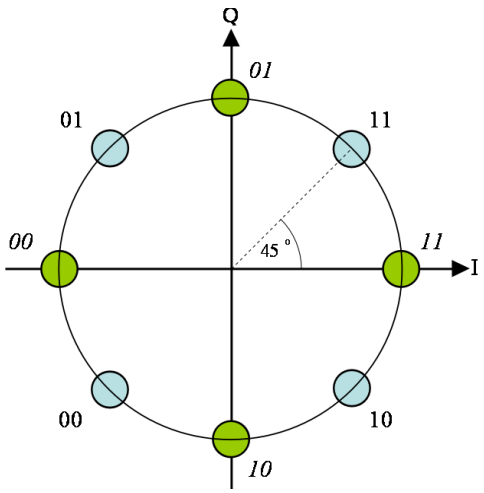

Figure 4: QPSK Constellation Diagram

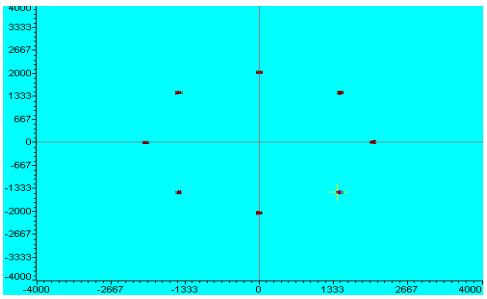

Figure 5: QPSK Constellation Diagram

Source Code Fixed Point Arithmetic

FIR Filter in Fixed Point arithmetic

```
void fir_filter(short signal[],
                int size)
{
    int acc;
    int prod;
    int i, j;

  for(j = 0; j < NUM_TAPS; ++j)
      R_in[j] = 0;

   for(j = 0; j < size; ++j)
    {
        /* Update most recent sample */
              R_in[0] =  signal[j] ;

          acc = 0;

          for (i = 0; i < NUM_TAPS; i++)
          {
     prod = (int)bcoeff[i] * R_in[i];
     acc += prod;
          }

          signal[j] = acc >> 15;

         /* Shift delay samples */
        for (i = NUM_TAPS-1; i > 0; i--)
         R_in[i]=R_in[i-1];
  }
}
```

$\pi/4$-DQPSK Modulator

```
/// Convert to Q15 format
inline short Q15(int a)
{
 if( a >= 0 )
    return  (a >> 15) ?  (a >> 15) : a;
 else
    return  (-a >> 15) ? -(-a >> 15) : a;
}

/*   SQRT(2) / 2   in Q15 format */
#define multFact  23170

/* Function "encodeQPSK"
**  Description:
**  encode message using
**  QPSDK pi/4 modulation
**
** INPUT :
**   message   the input message
**   size      the imput message size
** OUTPUT:
**  signal     the modulation signal
**
*/
extern void
encodeQPSK(short message[],
           short signal[],
           int size)
{
 int i, j, halfsize, Sample_Range;
 int oldI, oldQ, newI, newQ;
 int replicate_carrier;
 short tbc1, tbc2, t1, t2;

 halfsize     = size >> 1;
 Sample_Range = NUM_SAMPLES * halfsize;

 // Remap to {-1, 1}
 for(i = 0; i < size; ++i)
   message[i] = (message[i] << 1) - 1;

 oldI = 1;
 oldQ = 0;

 // Returns the phase shifts for pi/4 DQPSK
 // from the input binary data stream
 for(i = 0; i < halfsize; ++i)
 {
    tbc1 = message[i * 2];
    tbc2 = message[i * 2 + 1];

    newI = Q15((oldI*tbc1 - oldQ*tbc2) *
               multFact );

    newQ = Q15((oldQ*tbc1 + oldI*tbc2) *
               multFact );

    oldI = newI;
    oldQ = newQ;

    Ik[i] = newI;
    Qk[i] = newQ;
  }

 j = 0;

 for(i = 0; i < Sample_Range; ++i)
  if( i % NUM_SAMPLES == 0 )
   {
     Isam[i] = Ik[j];
     Qsam[i] = Qk[j];
     j = j + 1;
   }
```

```
  else
     Isam[i] = Qsam[i] = 0;

 fir_filter( Isam, Sample_Range );
 fir_filter( Qsam, Sample_Range );

 // Upmix with carrier
 for(j = 0; j < Sample_Range; ++j)
 {
   replicate_carrier = j % CARRIER_SIZE;
   t1 = Q15(Isam[j] *
            Icarrier[replicate_carrier]);

   t2 = Q15(Qsam[j] *
       Qcarrier[replicate_carrier]);

   signal[j] =  t1 + t2;
 }
}
```

$\pi/4$-DQPSK Demodulator

```
/* Function "decodeQPSK"
**  Description:
**  decode signal using QPSDK pi/4
**
** INPUT :
**  signal  the modulation signal
**  size    signal size
** OUTPUT:
**  message  the transmitted message
**
*/
extern void
decodeQPSK(short signal[],
           short message[],
                 int size)
{
 int i, j, replicate_carrier;

 for(j = 0; j < size; ++j)
 {
    replicate_carrier = j % CARRIER_SIZE;

    Irec[j] = Q15(signal[j] *
        Icarrier[replicate_carrier]);

    Qrec[j] = Q15(signal[j] *
        Qcarrier[replicate_carrier]);
 }

 fir_filter(Irec, size);
 fir_filter(Qrec, size);

 i = 0;
 for(j = NUM_TAPS; j < size; j +=NUM_SAMPLES )
 {
  Ik[i] = Irec[j];
  Qk[i] = Qrec[j];
  ++i;
 }

 // Loop through and decode the phases
 for(j = 1; j < HALF_BUFF_SIZE; ++j)
  {
    message[(j << 1) - 2] =
  (Qk[j] * Qk[j-1] + Ik[j] * Ik[j-1]) > 0;

    message[(j << 1) - 1] =
        (Ik[j-1] * Qk[j] - Ik[j] * Qk[j-1]) > 0;
  }
}
```