FEC Decoding

By Siri Johnsrud and Robin Hoel

Keywords

- *FEC*
- *Trellis*
- *Viterbi*
- *CC1100*
- *CC1100E*
- *CC1101*
- *CC1110Fx*
- *CC1111Fx*
- *CC1150*
- *CC2500*
- *CC2510Fx*
- *CC2511Fx*
- *CC2550*
- *CC430Fx*

1 Introduction

The CC1100, CC1100E, CC1101, CC1110Fx, CC1111Fx, CC1150, CC2500, CC2510Fx, CC2511Fx, and CC2550 all implement FEC encoding and decoding in HW. The purpose of this design note is to describe how one can implement the same FEC decoding in SW. This is in particular very important for the CC430Fx device, which has the same radio as the CC1101 and CC1110/11Fx but without HW FEC included. This design note is not meant as a tutorial on FEC and it will not cover the FEC encoding, as it is described in DN504 [1].

Table of Contents

KEYWORDS **1**
1 INTRODUCTION **1**
2 ABBREVIATIONS **2**
3 IMPLEMENTATION **3**
3.1 CODE EXAMPLE ASSUMPTIONS AND LIMITATIONS 3
3.2 C CODE 4
4 EXPLANATION TO THE CODE **8**
5 REFERENCES **16**
6 GENERAL INFORMATION **17**
6.1 DOCUMENT HISTORY 17

2 Abbreviations

CRC	Cyclic Redundancy Check
FEC	Forward Error Correction
FIFO	First In First Out
HW	Hardware
LSB	Least Significant Bit
MSB	Most Significant Bit
RAM	Random Access Memory
SW	Software

3 Implementation

3.1 Code Example Assumptions and Limitations

Assume that you want to use the CC1101 [5] to transmit a packet and the CC430Fx [2] to receive it. The payload is 29 bytes and 2 bytes of CRC are appended. If FEC is enabled (`MDMCFG1.FEC_EN = 1`) on the transmitter, 64 bytes will be transmitted over the air in addition to preamble and sync word. Due to the appended trellis terminator and the size of the interleaving buffer the packet length will always be a multiple of 4 after encoding (see DN504 [1]). The number of bytes sent over the air (not including preamble and sync word) can be calculated as shown in Equation 1.

$$\#\text{of Bytes on the Air} = \left(\left(\frac{\text{Payload Length} + 2\ \text{Bytes Optional CRC}}{2} \right) + 1 \right) \cdot 4$$

Equation 1. # of Bytes on the Air[1]

The code example shown in Section 3.2 does not show how to set up the CC430Fx to receive a packet and it does not show how to implement the function (***readRxFifo***) that will read from the RXFIFO in Figure 5 (see the CC430Fx User's Guide [2] on how this can be done). It is assumed that a flag, ***packetReceived***, is asserted when the packet is received and the 64 bytes are in the RXFIFO. It is important to notice that there is no packet size limitation to the FEC decoding itself, but if more than 64 bytes are sent over the air, the receiver must start to read the RXFIFO before the complete packet is received and additional RAM is needed to store the un-coded packet. When 64 bytes or less is sent on the air, the un-coded data can simply be kept in the RXFIFO until being decoded. The CC1101 [5] only support fixed packet length mode (`PKTCTRL0.LENGTH_CONFIG = 0`) when FEC is enabled, so the CC430Fx should also use this mode. Overflow of the RXFIFO will therefore not be an issue as long as the maximum packet length is less than 64 bytes (`PKTLEN <= 0x40`).

If the receiver is not the CC430Fx but some other radio which do not have an RXFIFO, it will be necessary to store the receive packet in a temporary buffer and then the function ***readRxFifo*** should read from this buffer instead of from the RXFIFO. It is assumed that the bytes in this buffer are read in the same order as they would have been read from a traditional FIFO.

[1] The division is a "whole number" division; i.e., all variables are of type ***unsigned short***

3.2 C Code

The C code in this section is organized as follows: Figure 1; Function Prototypes, Global Variables, and Defines, Figure 2; Function Definitions, Figure 3 and Figure 4; FEC Decoder Implementation, and Figure 5; main.

```
/**************************************************************************************************
* FUNCTION PROTOTYPES
*/
unsigned short fecDecode(unsigned char *pDecData, unsigned char* pInData, unsigned short RemBytes);
static unsigned char hammWeight(unsigned char a);
static unsigned char min(unsigned char a, unsigned char b);
static unsigned short calcCRC(unsigned char crcData, unsigned short crcReg);


/**************************************************************************************************
* GLOBAL VARIABLES
*/

// The payload + CRC are 31 bytes. This way the complete packet to be received will fit in the RXFIFO
unsigned char rxBuffer[4];    // Buffer used to hold data read from the RXFIFO (4 bytes are read at a time)
unsigned char rxPacket[31];   // Data + CRC after being interleaved and decoded

// Look-up source state index when:
//                      Destination state --\    /-- Each of two possible source states
const unsigned char aTrellisSourceStateLut[8][2] =
{
  {0, 4},      // State {0,4} -> State 0
  {0, 4},      // State {0,4} -> State 1
  {1, 5},      // State {1,5} -> State 2
  {1, 5},      // State {1,5} -> State 3
  {2, 6},      // State {2,6} -> State 4
  {2, 6},      // State {2,6} -> State 5
  {3, 7},      // State {3,7} -> State 6
  {3, 7},      // State {3,7} -> State 7
};

// Look-up expected output when:
//                        Destination state --\    /-- Each of two possible source states
const unsigned char aTrellisTransitionOutput[8][2] =
{
  {0, 3},      // State {0,4} -> State 0 produces {"00", "11"}
  {3, 0},      // State {0,4} -> State 1 produces {"11", "00"}
  {1, 2},      // State {1,5} -> State 2 produces {"01", "10"}
  {2, 1},      // State {1,5} -> State 3 produces {"10", "01"}
  {3, 0},      // State {2,6} -> State 4 produces {"11", "00"}
  {0, 3},      // State {2,6} -> State 5 produces {"00", "11"}
  {2, 1},      // State {3,7} -> State 6 produces {"10", "01"}
  {1, 2},      // State {3,7} -> State 7 produces {"01", "10"}
};

// Look-up input bit at encoder when:
//                       Destination state --\
const unsigned char aTrellisTransitionInput[8] =
{
  0,
  1,
  0,
  1,
  0,
  1,
  0,
  1,
};

/**************************************************************************************************
* DEFINES
*/

// NUMBER_OF_BYTES_AFTER_DECODING should be given the length of the payload + CRC (CRC is optional)
#define NUMBER_OF_BYTES_AFTER_DECODING  31
#define NUMBER_OF_BYTES_BEFORE_DECODING (4 * ((NUMBER_OF_BYTES_AFTER_DECODING / 2) + 1))
```

Figure 1. Function Prototypes, Global Variables, and Defines

```
/***********************************************************************************************************
* @fn          hammWeight
*
* @brief       Calculates Hamming weight of byte (# bits set)
*
* @param       a - Byte to find the Hamming weight for
*
* @return      Hamming weight (# of bits set in a)
*/
static unsigned char hammWeight(unsigned char a)
{
  a = ((a & 0xAA) >> 1) + (a & 0x55);
  a = ((a & 0xCC) >> 2) + (a & 0x33);
  a = ((a & 0xF0) >> 4) + (a & 0x0F);
  return a;
}

/***********************************************************************************************************
* @fn          min
*
* @brief       Returns the minimum of two values
*
* @param       a - Value 1
*              b - Value 2
*
* @return      Minimum of two values
*              Value 1 (Value 1 < Value 2)
*              Value 2 (Value 2 < Value 1)
*/
static unsigned char min(unsigned char a, unsigned char b)
{
  return (a <= b ? a : b);
}

/***********************************************************************************************************
* @fn          calcCRC
*
* @brief       Calculates a checksum over n data bytes
*              Example of usage
*
*              checksum = 0xFFFF;
*              for (i = 0; i < n; i++)
*                 checksum = calcCRC(dataBytes[i], checksum);
*
* @param       crcData - checksum (initially set to 0xFFFF)
*              crcReg - data byte
*
*
* @return      Checksum
*/
static unsigned short calcCRC(unsigned char crcData, unsigned short crcReg)
{
  unsigned char i;
  for (i = 0; i < 8; i++) {
    if (((crcReg & 0x8000) >> 8) ^ (crcData & 0x80))
      crcReg = (crcReg << 1) ^ 0x8005;
    else
      crcReg = (crcReg << 1);
    crcData <<= 1;
  }
  return crcReg;
}
```

Figure 2. Function Definitions

```
/***********************************************************************************************************
* @fn          fecDecode
*
* @brief       De-interleaves and decodes a given input buffer
*
* @param       pDecData  - Pointer to where to put decoded data (NULL when initializing at start of packet)
*              pInData   - Pointer to received data
*              nRemBytes - of remaining (decoded) bytes to decode
*
*
* @return      Number of bytes of decoded data stored at pDecData
*/
unsigned short fecDecode(unsigned char *pDecData, unsigned char* pInData, unsigned short nRemBytes)
{
  // Two sets of buffers (last, current) for each destination state for holding:
  static unsigned char nCost[2][8];           // Accumulated path cost
  static unsigned long aPath[2][8];           // Encoder input data (32b window)

  // Indices of (last, current) buffer for each iteration
  static unsigned char iLastBuf;
  static unsigned char iCurrBuf;

  // Number of bits in each path buffer
  static unsigned char nPathBits;

  // Variables used to hold # Viterbi iterations to run, # bytes output,
  // minimum cost for any destination state, bit index of input symbol
  unsigned char nIterations;
  unsigned short nOutputBytes = 0;
  unsigned char nMinCost;
  signed char iBit = 8 - 2;

  // Initialize variables at start of packet (and return without doing any more)
  if (pDecData == NULL) {
    unsigned char n ;
    memset(nCost, 0, sizeof(nCost));
    for (n = 1; n < 8; n++)
      nCost[0][n] = 100;
    iLastBuf = 0;
    iCurrBuf = 1;
    nPathBits = 0;
    return 0;
  }

  {
    unsigned char aDeintData[4];
    signed char iOut;
    signed char iIn;

    // De-interleave received data (and change pInData to point to de-interleaved data)
    for (iOut = 0; iOut < 4; iOut++) {
      unsigned char dataByte = 0;
      for (iIn = 3; iIn >= 0; iIn--)
        dataByte = (dataByte << 2) | ((pInData[iIn] >>( 2 * iOut)) & 0x03);
      aDeintData[iOut] = dataByte;
    }
    pInData = aDeintData;
  }

  // Process up to 4 bytes of de-interleaved input data, processing one encoder symbol (2b) at a time
  for (nIterations = 16; nIterations > 0; nIterations--) {

    unsigned char iDestState;
    unsigned char symbol  = ((*pInData) >> iBit) & 0x03;

    // Find minimum cost so that we can normalize costs (only last iteration used)
    nMinCost = 0xFF;

    // Get 2b input symbol (MSB first) and do one iteration of Viterbi decoding
    if ((iBit -= 2) < 0) {
      iBit = 6;
      pInData++;            // Update pointer to the next byte of received data
    }

    // For each destination state in the trellis, calculate hamming costs for both possible paths into state and
    // select the one with lowest cost.
    for (iDestState = 0; iDestState < 8; iDestState++) {
      unsigned char nCost0;
      unsigned char nCost1;
      unsigned char iSrcState0;
      unsigned char iSrcState1;
      unsigned char nInputBit;

      nInputBit = aTrellisTransitionInput[iDestState];

      // Calculate cost of transition from each of the two source states (cost is Hamming difference between
      // received 2b symbol and expected symbol for transition)
      iSrcState0 = aTrellisSourceStateLut[iDestState][0];
      nCost0   = nCost[iLastBuf][iSrcState0];
      nCost0 += hammWeight(symbol ^ aTrellisTransitionOutput[iDestState][0]);

      iSrcState1 = aTrellisSourceStateLut[iDestState][1];
      nCost1   = nCost[iLastBuf][iSrcState1];
      nCost1 += hammWeight(symbol ^ aTrellisTransitionOutput[iDestState][1]);
```

Figure 3. FEC Decoder Implementation (1)

```
      // Select transition that gives lowest cost in destination state, copy that source state's path and add
      // new decoded bit
      if (nCost0 <= nCost1) {
        nCost[iCurrBuf][iDestState] = nCost0;
        nMinCost = min(nMinCost, nCost0);
        aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState0] << 1) | nInputBit;
      } else {
        nCost[iCurrBuf][iDestState] = nCost1;
        nMinCost = min(nMinCost, nCost1);
        aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState1] << 1) | nInputBit;
      }
    }
    nPathBits++;

    // If trellis history is sufficiently long, output a byte of decoded data
    if (nPathBits == 32) {
      *pDecData++ = (aPath[iCurrBuf][0] >> 24) & 0xFF;
      nOutputBytes++;
      nPathBits -= 8;
      nRemBytes--;
    }

    // After having processed 3-symbol trellis terminator, flush out remaining data
    if ((nRemBytes <= 3) && (nPathBits == ((8 * nRemBytes) + 3))) {
      while (nPathBits >= 8) {
        *pDecData++ = (aPath[iCurrBuf][0] >> (nPathBits - 8)) & 0xFF;
        nOutputBytes++;
        nPathBits -= 8;
      }
      return nOutputBytes;
    }

    // Swap current and last buffers for next iteration
    iLastBuf = (iLastBuf + 1) % 2;
    iCurrBuf = (iCurrBuf + 1) % 2;
  }

  // Normalize costs so that minimum cost becomes 0
  {
    unsigned char iState;
    for (iState = 0; iState < 8; iState++)
      nCost[iLastBuf][iState] -= nMinCost;
  }
  return nOutputBytes;
}
```

Figure 4. FEC Decoder Implementation (2)

```
/*******************************************************************************************************************
* @fn          main
*
* @brief       This code example demonstrates how the fecDecode function can be used. It is assumed that a
*              flag, packetReceived, is asserted when a packet is received (there are 64 bytes in the RXFIFO)
*
* @param       None
*
* @return      None
*/
void main(void)
{
  unsigned short checksum;
  unsigned short nBytes;
  unsigned char *pDecData = rxPacket;           // Destination for decoded data

  // Init MCU and Radio

  while (1) {

    while (!packetReceived);                    // Wait for packet to be received (64 bytes in the RXFIFO)
    packetReceived = 0;

    pDecData = rxPacket;

    // Perform de-interleaving and decoding (both done in the same function)
    fecDecode(NULL, NULL, 0);                   // The function needs to be called with a NULL pointer for
                                                // initialization before every packet to decode
    nBytes = NUMBER_OF_BYTES_AFTER_DECODING;
    while (nBytes > 0) {
      unsigned short nBytesOut;
      readRxFifo(RF_RXFIFO, rxBuffer, 4);       // Read 4 bytes from the RXFIFO and store them in rxBuffer
      nBytesOut = fecDecode(pDecData, rxBuffer, nBytes);
      nBytes -= nBytesOut;
      pDecData += nBytesOut;
    }

    // Perform CRC check (Optional)
    {
      unsigned short i;
      nBytes = NUMBER_OF_BYTES_AFTER_DECODING;
      checksum = 0xFFFF;                        // Init value for CRC calculation
      for (i = 0; i < nBytes; i++)
        checksum = calcCRC(rxPacket[i], checksum);
      if (!checksum) {
        // Do something to indicate that the CRC is OK
      }
    }
  }
}
```

Figure 5. main

4 Explanation to the Code

The most important part of the code is the decoder part implemented in the function ***fecDecode*** (see Figure 3 and Figure 4). The function will process 4 and 4 bytes of received data since this is the data size the interleaver works on. This means that in most cases the function will be called several times for each received packet. The pseudo code for the function is shown in Figure 6.

```
fecDecode()
{
  // Variable Declaration

  // Initialize variables at start of packet (and return without doing any more)

  // De-interleave 4 bytes of received data (4 bytes of data means 16 (2b) encode symbols

  // For all 16 symbols do one iteration of Viterbi decoding
  for (nIterations = 16; nIterations > 0; nIterations--) {

    // Get 2b input symbol (MSB first) and do one iteration of Viterbi decoding

    // For each destination state in the trellis
    for (iDestState = 0; iDestState < 8; iDestState++) {

      // Calculate cost of transition from each of the two source states (cost is Hamming difference between
      // received 2b symbol and expected symbol for transition)

      // Select transition that gives lowest cost in destination state, copy that source state's path and add
      // new decoded bit
    }
    nPathBits++;

    // If trellis history is sufficiently long, output a byte of decoded data. After 32 iterations
    // (nPathBits == 32) the 8 MSB bits will be the same for all 8 surviving paths and a byte can be output

    // After having processed 3-symbol trellis terminator, flush out remaining data

    // Swap current and last buffers for next iteration
  }
  // Normalize costs so that minimum cost becomes 0
}
```

Figure 6. Pseudo Code for the FEC Encoder

The “key” elements of the code are the two for loops that for every symbol goes through each destination state in the trellis. An example is used to illustrate what is going on in this loop.

Example: A packet consisting of 5 bytes (0x01, 0x02, 0x03, 0x04, 0x05) is being interleaved and encoded by the CC1101 (`MDMCFG1.FEC_EN = 1`). The data transmitted on the air will be the following (preamble and sync word is not shown):

0x4C, 0xF0, 0x30, 0x10, 0xC8, 0x7C, 0xC3, 0x23, 0x40, 0x34, 0x7C, 0xE3 (see DN504 [1])

In chunks of 4 and 4 bytes, this data will on the receiver side be interleaved, giving the symbols shown in Table 1 to be decoded (only the 4 first bytes are shown):

Symbol #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Symbol	00	00	00	00	00	00	00	11	01	11	11	00	00	00	11	01
Byte	0x00				0x03				0x7C				0x0D			

Table 1. Symbols to be Encoded

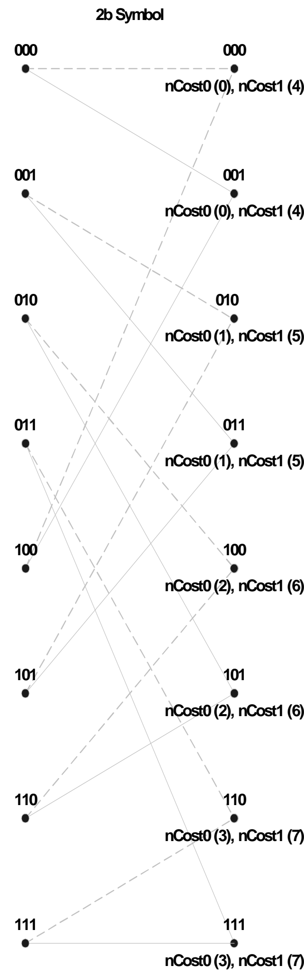

Figure 7. Trellis 1 (the number in () indicates the source state of that given cost)

For each received symbol (2b), all possible encoder output symbols (00, 01, 10, and 11) in Figure 7 are compared against the received symbol and a transition cost is calculated (nCost0 and nCost1). The appropriate transition cost is added to the accumulated path cost of each path that terminates in the source state on the left in the figure. It can be seen that there are two transitions into each destination state on the right in the figure. For each destination state the incoming transition with the lowest accumulated path cost is selected (the survivor path) and the other one thrown away - nothing is lost as all future paths that go through this state at this point in the trellis would do the same selection. Thus the number of paths that the Viterbi Algorithm tracks is always constant and the optimal path is always one of them.

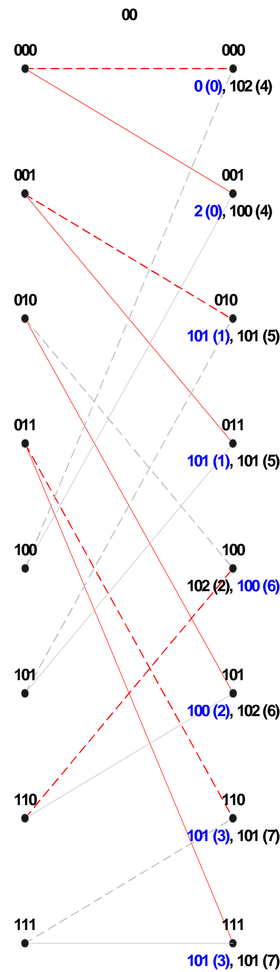

Figure 8. Trellis 2

```
iCurrBuf = 1
iLastBuf = 0

nCost[iCurrBuf][] = [0, 2, 101, 101, 100, 100, 101, 101]

aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[1][0] = (aPath[0][0] << 1) | Input to state 0 = 00000000000000000000000000000000
aPath[1][1] = (aPath[0][0] << 1) | Input to state 1 = 00000000000000000000000000000001
aPath[1][2] = (aPath[0][1] << 1) | Input to state 2 = 00000000000000000000000000000000
aPath[1][3] = (aPath[0][1] << 1) | Input to state 3 = 00000000000000000000000000000001
aPath[1][4] = (aPath[0][6] << 1) | Input to state 4 = 00000000000000000000000000000000
aPath[1][5] = (aPath[0][2] << 1) | Input to state 5 = 00000000000000000000000000000001
aPath[1][6] = (aPath[0][3] << 1) | Input to state 6 = 00000000000000000000000000000000
aPath[1][7] = (aPath[0][3] << 1) | Input to state 7 = 00000000000000000000000000000001
```

Figure 9. *nCost* and *aPath* after 1st Iteration (received symbol: 00_b)

Figure 10. Trellis 3

```
iCurrBuf = 0
iLastBuf = 1

nCost[iCurrBuf][] = [0, 2, 3, 3, 101, 101, 102, 102]

aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[0][0] = (aPath[1][0] << 1) | Input to state 0 = 00000000000000000000000000000000
aPath[0][1] = (aPath[1][0] << 1) | Input to state 1 = 00000000000000000000000000000001
aPath[0][2] = (aPath[1][1] << 1) | Input to state 2 = 00000000000000000000000000000010
aPath[0][3] = (aPath[1][1] << 1) | Input to state 3 = 00000000000000000000000000000011
aPath[0][4] = (aPath[1][6] << 1) | Input to state 4 = 00000000000000000000000000000000
aPath[0][5] = (aPath[1][2] << 1) | Input to state 5 = 00000000000000000000000000000001
aPath[0][6] = (aPath[1][3] << 1) | Input to state 6 = 00000000000000000000000000000010
aPath[0][7] = (aPath[1][3] << 1) | Input to state 7 = 00000000000000000000000000000011
```

Figure 11. *nCost* and *aPath* after 2nd Iteration (received symbol: 00_b)

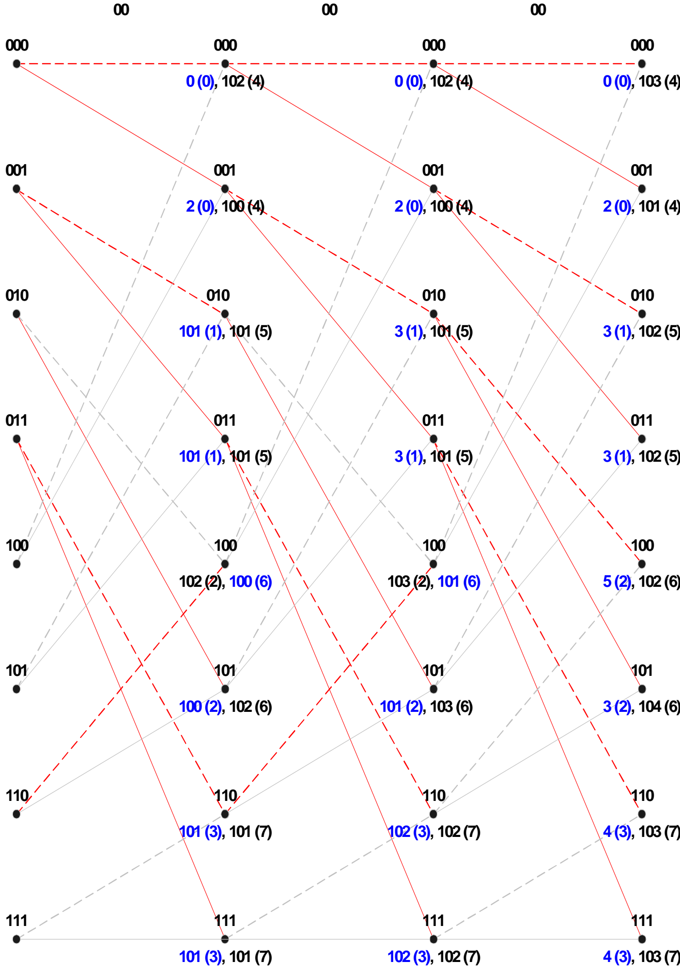

Figure 12. Trellis 4

```
iCurrBuf = 1
iLastBuf = 0

nCost[iCurrBuf][] = [0, 2, 3, 3, 5, 3, 4, 4]

aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[1][0] = (aPath[0][0] << 1) | Input to state 0 = 00000000000000000000000000000000
aPath[1][1] = (aPath[0][0] << 1) | Input to state 1 = 00000000000000000000000000000001
aPath[1][2] = (aPath[0][1] << 1) | Input to state 2 = 00000000000000000000000000000010
aPath[1][3] = (aPath[0][1] << 1) | Input to state 3 = 00000000000000000000000000000011
aPath[1][4] = (aPath[0][2] << 1) | Input to state 4 = 00000000000000000000000000000100
aPath[1][5] = (aPath[0][2] << 1) | Input to state 5 = 00000000000000000000000000000101
aPath[1][6] = (aPath[0][3] << 1) | Input to state 6 = 00000000000000000000000000000110
aPath[1][7] = (aPath[0][3] << 1) | Input to state 7 = 00000000000000000000000000000111
```

Figure 13. *nCost* and *aPath* after 3rd Iteration (received symbol: 00_b)

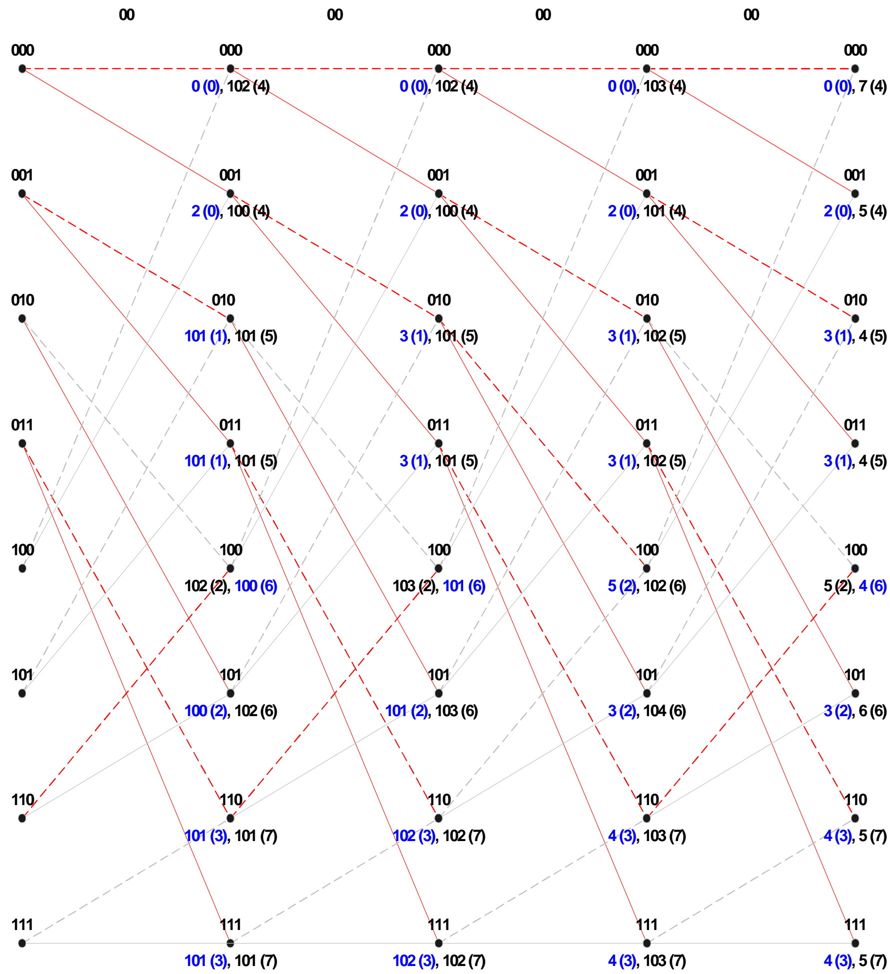

Figure 14. Trellis 5

```
iCurrBuf = 0
iLastBuf = 1

nCost[iCurrBuf][] = [0, 2, 3, 3, 4, 3, 4, 4]

aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[0][0] = (aPath[1][0] << 1) | Input to state 0 = 00000000000000000000000000000000
aPath[0][1] = (aPath[1][0] << 1) | Input to state 1 = 00000000000000000000000000000001
aPath[0][2] = (aPath[1][1] << 1) | Input to state 2 = 00000000000000000000000000000010
aPath[0][3] = (aPath[1][1] << 1) | Input to state 3 = 00000000000000000000000000000011
aPath[0][4] = (aPath[1][6] << 1) | Input to state 4 = 00000000000000000000000000001100
aPath[0][5] = (aPath[1][2] << 1) | Input to state 5 = 00000000000000000000000000000101
aPath[0][6] = (aPath[1][3] << 1) | Input to state 6 = 00000000000000000000000000000110
aPath[0][7] = (aPath[1][3] << 1) | Input to state 7 = 00000000000000000000000000000111
```

Figure 15. *nCost* and *aPath* after 4th Iteration (received symbol: 00$_b$)

Figure 16. Trellis 6

```
iCurrBuf = 1
iLastBuf = 0

nCost[iCurrBuf][] = [0, 2, 3, 3, 4, 3, 4, 4]

aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[1][0] = aPath[0][0] << 1 | Input to state 0 = 00000000000000000000000000000000
aPath[1][1] = aPath[0][0] << 1 | Input to state 1 = 00000000000000000000000000000001
aPath[1][2] = aPath[0][1] << 1 | Input to state 2 = 00000000000000000000000000000010
aPath[1][3] = aPath[0][1] << 1 | Input to state 3 = 00000000000000000000000000000011
aPath[1][4] = aPath[0][6] << 1 | Input to state 4 = 00000000000000000000000000001100
aPath[1][5] = aPath[0][2] << 1 | Input to state 5 = 00000000000000000000000000000101
aPath[1][6] = aPath[0][3] << 1 | Input to state 6 = 00000000000000000000000000000110
aPath[1][7] = aPath[0][3] << 1 | Input to state 7 = 00000000000000000000000000000111
```

Figure 17. *nCost* and *aPath* after 5st Iteration (received symbol: 00_b)

After having processed 32 symbol, the 8 MSBs of ***aPath[0][0]*** (0000001_b) is copied to ***rxFifo[0]***.

```
aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[0][0] = 00000001000000100000001100001000
aPath[0][1] = 00000001000000100000001100000001
aPath[0][2] = 00000001000000100000001100000010
aPath[0][3] = 00000001000000100000001100000011
aPath[0][4] = 00000001000000100000001100000100
aPath[0][5] = 00000001000000100000001100000101
aPath[0][6] = 00000001000000100000001100000110
aPath[0][7] = 00000001000000100000001100000111
```

Figure 18. *aPath* after having Processed 32 Symbols

After 8 more symbols, ***aPath[iCurrBuf][iDestState]*** looks like in Figure 19 and 00000010_b (8 MSBs of ***aPath[0][0]***) are copied to ***rxFifo[1]***.

```
aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[0][0] = 00000010000000110000010000000000
aPath[0][1] = 00000010000000110000010000001001
aPath[0][2] = 00000010000000110000010000000010
aPath[0][3] = 00000010000000110000010000000011
aPath[0][4] = 00000010000000110000010000000100
aPath[0][5] = 00000010000000110000010000000101
aPath[0][6] = 00000010000000110000010000000110
aPath[0][7] = 00000010000000110000010000000111
```

Figure 19. *aPath* after having Processed 40 Symbols

After having processed 3 more symbols, the trellis terminator has been processed and the remaining bytes of the packet are being copied to ***rxFifo***.

rxFifo[2] = ***aPath[1][0]*** bit 26:19 = 00000011_b
rxFifo[3] = ***aPath[1][0]*** bit 18:11 = 00000100_b
rxFifo[4] = ***aPath[1][0]*** bit 10:3 = 00000101_b

```
aPath[iCurrBuf][iDestState] = (aPath[iLastBuf][iSrcState] << 1) | nInputBit;
aPath[1][0] = 00010000000110000010000000101000
aPath[1][1] = 00010000000110000010000000101001
aPath[1][2] = 00010000000110000010000000101010
aPath[1][3] = 00010000000110000010000000101011
aPath[1][4] = 00010000000110000010000000100100
aPath[1][5] = 00010000000110000010000000101101
aPath[1][6] = 00010000000110000010000000101110
aPath[1][7] = 00010000000110000010000000101111
```

Figure 20. *aPath* after having Processed 43 Symbols

5 References

[1] DN504 FEC Implementation (swra113.pdf)

[2] CC430 User's Guide (slau259.pdf)

[3] CC1100 Single-Chip Low Cost Low Power RF-Transceiver, Data sheet (cc1100.pdf)

[4] CC1100E Low-Power Sub-GHz RF Transceiver (470-510 MHz & 950-960 MHz) (CC1100E.pdf)

[5] CC1101 Single-Chip Low Cost Low Power RF-Transceiver, Data sheet (cc1101.pdf)

[6] CC1110Fx/CC1111Fx Low-Power Sub-1 GHz RF System-on-Chip (SoC) with MCU, Memory, Transceiver, and USB Controller (cc1110f32.pdf)

[7] CC1150 Single Chip Low Cost Low Power RF-Transmitter (cc1150.pdf)

[8] CC2500 Single-Chip Low Cost Low Power RF-Transceiver, Data sheet (cc2500.pdf)

[9] CC2510Fx/CC2511Fx Low-Power SoC (System-on-Chip) with MCU, Memory, 2.4 GHz RF Transceiver, and USB Controller (cc2510f32.pdf)

[10] CC2550 Low-Cost Low-Power 2.4 GHz RF Transmitter (cc2550.pdf)

6 General Information

6.1 Document History

Revision	Date	Description/Changes
SWRA313	2010.01.25	Initial release.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2010, Texas Instruments Incorporated