AN2835
Rev. 0, 9/2004

Building a Convolutional Encoder Using RCF Technology

by Wim Rouwet

Convolutional encoding is a forward error correcting (FEC) process associated with a Viterbi decoder on the receive side. Adding redundancy to the input data before it is sent to the channel enables it to recover from errors introduced in the channel. Typical applications for convolutional encoding are global system for mobile communications (GSM), wideband code division multiple access (WCDMA) symbol rate, and WiMAX. This application note describes how to design and implement a WiMAX convolutional encoder using RCF technology. It shows how the encoder mathematically represents a polynomial multiplication that makes it especially effective on the parallel architecture of the RCF MRC6011 processor and greatly boosts performance in comparison to traditional implementations using shift registers. This document assumes that you have experience in building and debugging an application on the RCF MRC6011 processor.

CONTENTS

1 Basics of Convolutional Encoding

Convolutional encoding is a way to encode digital data before it is transmitted through a noisy (error-prone) channel. Convolutionally encoded data is usually decoded via the Viterbi algorithm implemented in either hardware or software. During encoding, *k* input bits are mapped to *n* output bits to give a rate *k/n* coded bit stream. The encoder consists of a shift register of *kL* stages, where *L* is the constraint length of the code. For the encoder depicted in **Figure 1**, *D* is a delay unit. *g0* and *g1* are the generator polynomials:

- $g0 = input_{t=0}\ (xor)\ input_{t=-1}\ (xor)\ input_{t=-2} = 1 + D + D^2$
- $g1 = input_{t=0}\ (xor)\ input_{t=-2} = 1 + D^2$

For each input bit, the encoder produces two output bits (*g0g1*), so it is a half-rate encoder.

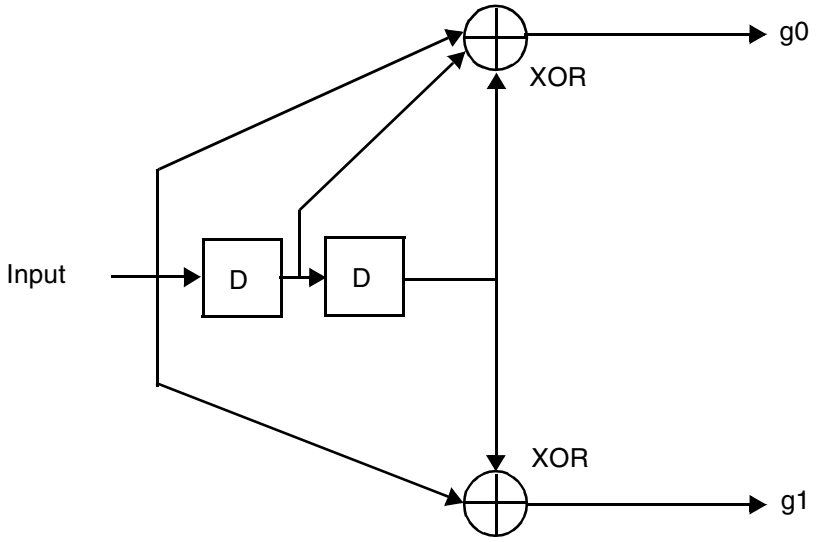


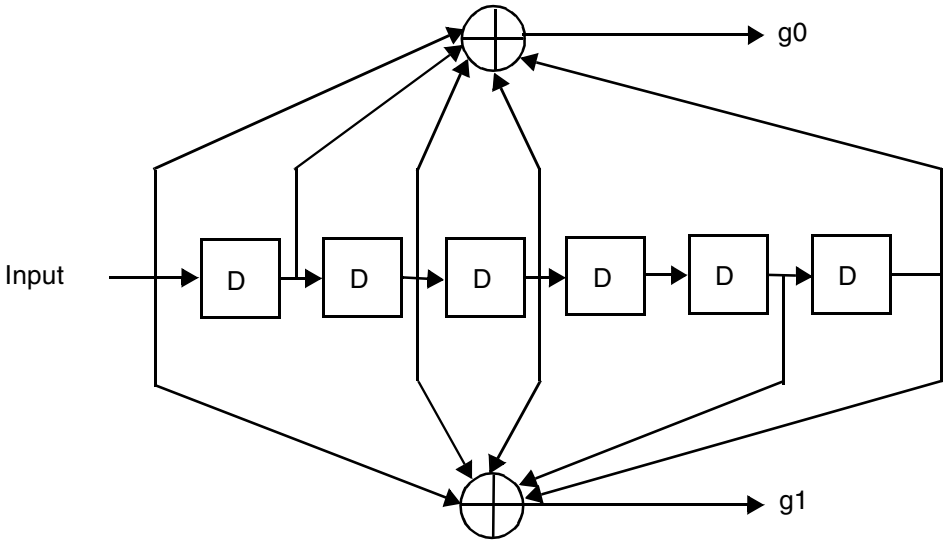
Figure 1. Simple Convolutional Encoder

Table 1 shows an example encoder input and the corresponding output in which $g0 = D^0 + D^1 + D^2$ and $g1 = D^0 + D^2$. The + indicates a logical XOR; the input sequence is 01001110, and the corresponding output is 0b0011101111011001.

Table 1. Encoder Input and Corresponding Output

Input	D^1	D^2	g0	g1	Output
0	0	0	0	0	00
1	0	0	1	1	11
0	1	0	1	0	10
0	0	1	1	1	11
1	0	0	1	1	11
1	1	0	0	1	01
1	1	1	1	0	10
0	1	1	0	1	01
	0	1			
		0			

Revision D5 of the IEEE WiMAX 802.16d standard yields a more complex convolutional encoder with a constraint length of 7 (see **Figure 2**). The generator polynomials for this encoder are $g0 = 171_{oct}$ and $g1 = 133_{oct}$. The encoder can easily be implemented in hardware shift registers. However, implementation in software can be a challenge if performance is of major importance.



From http://grouper.ieee.org/groups/802/16/

Figure 2. WiMAX Convolutional Encoder

To implement a high-performance convolutional encoder in software, especially on a parallel architecture, an understanding of the underlying mathematics is crucial. The first step is to represent the input bit string as a polynomial. Any sequence of 0's and 1's can be represented as a binary number or a polynomial. The convolutional encoder for WiMAX ($g0 = 171_{oct}$ and $g1 = 133_{oct}$) can be represented as follows:

- $g0 = 1 + D + D^2 + D^3 + D^6$
- $g1 = 1 + D^2 + D^3 + D^5 + D^6$

The convolutional encoder basically multiplies the generator polynomials by the input bit string, as follows:

- $A(x) = g0(x) * I(x) = a\ b\ c\ \ldots\ g$
- $B(x) = g1(x) * I(x) = P\ Q\ R\ \ldots\ V$

Interleaving the two outputs from the convolutional encoder yields $E(x) = aPbQcR\ \ldots\ gV$, which can also be written as:

$$E(x) = (a0\ b0\ c0\ \ldots\ g0) + (0P0Q0R\ \ldots\ 0V) = A(x^2) + x*B(x^2)$$

Therefore, $E(x) = A(x^2) + x * B(x^2)$ and $A(x^2) = g0(x^2) * I(x^2)$ *and* $B(x^2) = g1(x^2) * I(x^2)$, with the following:

$$E(x) = g0(x^2) * I(x^2) + x * g1(x^2) * I(x^2)$$

$$= I(x^2) * (g0(x^2) + x * g1(x^2))$$

$$= I(x^2) * G(x) \text{ where}$$

$$G(x) = g0(x^2) + x * g1(x^2)$$

$$G(x) = 1 + x^2 + x^4 + x^6 + x^{12} + x * (1 + x^4 + x^6 + x^{10} + x^{13}) =$$

$$G(x) = 1 + x + x^2 + x^4 + x^5 + x^6 + x^7 + x^{11} + x^{12} + x^{13}$$

To illustrate the polynomial multiplication, we work out an example input of 0x49 in three different ways:

- A classic hardware implementation using a shift register.
- Two individual polynomial multiplications with *g0* and *g1*.
- A polynomial multiplication of *G(x)*. Remember, in a polynomial multiplication, the + represents a logical XOR.

The input 0x49[1] in binary form is 0100 1001 (first bit to be encoded to the left, as common in the literature) or in polynomial form: $I(x) = x + x^4 + x^7$. In hardware, the encoding is as shown in **Table 2**.

Table 2. Hardware Encoding for the Input 0x49

Status of Convolutional Encoder							Output 0	Output 1
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	1	0
0	0	1	0	0	0	0	1	1
1	0	0	1	0	0	0	0	0
0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0
1	0	0	1	0	0	1	1	1
0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0
…	…							

The output sequence (first output 0, then output 1) corresponding to this input is 0011101100101011. This sequence is followed by a trail (‘1010…’) representing the memory of the encoder. The polynomial multiplication is as follows:

$$I(x) * g0(x) = (x + x^4 + x^7) * (1 + x + x^2 + x^3 + x^6)$$

$$= x + x^2 + x^3 + x^5 + x^6 + x^7 + x^8 + x^9 + x^{13}$$

$$= 0b01110111110001$$

$$I(x) * g1(x) = (x + x^4 + x^7) * (1 + x^2 + x^3 + x^5 + x^6)$$

$$= x + x^3 + x^7 + x^{12} + x^{13}$$

$$= 0b01010001000011$$

1. From the example in Revision D5 of the IEEE 802.16d specification, 8.3.3.5.1.

Interleaving $D(x) \times g0(x)$ *and* $D(x) \times g1(x)$ yields the output: 0b0011101100101011 101000000111. This output is the same as the output of the coding using the first method, including the trailing bits. Multiplication of $I(x^2)$ with $G(x)$ (method 3) gives the following output, as expected:

$$I(x^2) = x^2 + x^8 + x^{14}$$
$$I(x^2) \times G(x) = (x^2 + x^8 + x^{14}) * (1 + x + x^2 + x^4 + x^5 + x^6 + x^7 + x^{11} + x^{12} + x^{13})$$
$$= x^2 + x^3 + x^4 + x^6 + x^7 + x^{10} + x^{12} + x^{14} + x^{15} + x^{16} + x^{19} + x^{25} + x^{26} + x^{27}$$
$$= 0011101100101011\ 101000000111$$

The mathematics in this example prove that convolutional encoding is a simple and single polynomial multiplication, which we can use in designing an encoder. We perform the polynomial multiplication via a look-up table, storing the output of the convolutional encoder, corresponding to any input and input size, with the corresponding trail bits in the table. For example, a single-byte input (in the previous example, 0x49 or 0b0100 1001) gives an output of two bytes (0x3B2B or 0b0011 1011 0010 1011) and a total of 12 trail bits (6 delay slots, 2 bits per slot, 0xA07 or 1010 0000 0111). The linearity of the multiplication function proves that the trail bits can be added to the result of the next input byte (that is, the next look-up).

In another example for the WiMAX convolutional encoder, we encode the string 0x4931 40BF D4BA A112 byte-by-byte. The first step is to look up the polynomial multiplication for 0x49, which gives us 0x3B2B for the code and 0xA07 for the trail bytes. Next, we look up 0x31, which gives 0x0D4E and 0x0C70. XORing 0xA070 (notice the extra 0 added to align) and 0x0D4E yields the next output bytes, 0xAD3E. This process can be repeated to produce 0x37B7, 0xE288, 0xF1EC, 0x55B0, 0xBC2E, and 0x7FC2. The look-up table corresponding to this example is included in this application node as **Appendix A**. In an implementation using the MRC6011 RCF processor, we improve the coding speed by executing multiple look-ups in parallel.

2 RCF Convolutional Encoder Implementation

The implementation of the encoder on an RCF processor employs the parallel structure of the RCF computing array to execute multiple look-up tables in parallel. We must replicate the look-up table to multiple instances in frame buffer memory so that eight look-ups in parallel means eight replications of the table. Assuming a byte-by-byte look-up, each table entry consists of 16 bits of output data and 12 trail bits for a total of 28 bits. For memory alignment, we allocate 32 bits per entry, or two 16-bit entries for each possible input byte. The total memory consumption is $256 \times 32 \times 8 = 65536$ bits or 8 KB. The first step is to take the input data for the encoder to the array, one byte per column in the first row, as shown in the following code sample:

```
MORPHO {
                CELL{0,*} R1 = BYP{FB{mydata, 0x00, OMEGA_BR2, COL_BUS, WORD}};
                CELL{1,*} NOP{};}
        MORPHO {
                CELL{0,*} R2 = BYP{FB{mydata, 0x02, OMEGA_BR2, COL_BUS, WORD}};
                CELL{1,*} NOP{};}
        MORPHO {
                CELL{0,*} R3 = BYP{FB{mydata, 0x04, OMEGA_BR2, COL_BUS, WORD}};
                CELL{1,*} NOP{};}
        MORPHO {
                CELL{0,*} R4 = BYP{FB{mydata, 0x06, OMEGA_BR2, COL_BUS, WORD}};
                CELL{1,*} NOP{};}
        MORPHO {
                CELL{*,0}:R8 R8 = PACKH{R0,R1};
                CELL{*,1}:R8 R8 = PACKL{R0,R1};
                CELL{*,2}:R8 R8 = PACKH{R0,R2};
                CELL{*,3}:R8 R8 = PACKL{R0,R2};
                CELL{*,4}:R8 R8 = PACKH{R0,R3};
```

```
                        CELL{*,5}:R8 R8 = PACKL{R0,R3};
                        CELL{*,6}:R8 R8 = PACKH{R0,R4};
                        CELL{*,7}:R8 R8 = PACKL{R0,R4};}
                // At this point, R8 contains for all columns the low byte.
```

The input data is distributed over the array from 4×16 bits to 8×8 bits. The next step is to look up the output bytes for the contents of R8 (the index). The output bytes are stored in the first table (LUT1), and the trail bytes are stored in the second table (LUT2):

```
        stw LUT1, $0, 0xfffff210;
MORPHO {
                CELL{0,*} R1 = BYP{FB{CELL{0,*}, ALL_BANKS, OMEGA_RT, COL_BUS, WORD}};
                        CELL{1,*}:R0 NOP{};}
```

For the trail bytes, we use the index of the previous column, as shown in the pseudo code:

```
MORPHO {
                CELL{*,0}:R8 R9 = BYP{R8{*,$+7}};
                        CELL{*,1}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,2}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,3}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,4}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,5}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,6}:R8 R9 = BYP{R8{*,$-1}};
                        CELL{*,7}:R8 R9 = BYP{R8{*,$-1}};}
        stw LUT2, $0, 0xfffff210;
                MORPHO {
                        CELL{0,*} R2 = BYP{FB{CELL{0,*}, ALL_BANKS, OMEGA_RT, COL_BUS, WORD}};
                        CELL{1,*}:R0 NOP{};}
```

The two lookup results are XORed to get the final result:

```
                MORPHO {
                        CELL{0,*}:R7 R7 = XOR{R1,R2};
                        CELL{1,*}:R0  NOP{};}
```

Then the result can be written to the output memory section:

```
FB{output,0x0, CELL{0,*}} = CELL{*,*}:OUT_REG NOP{};
```

The complete code for the WiMAX convolutional encoder using the pipeline and including setup code is as follows:

```
MORPHO_ASM(mydata, LUT1, LUT2, output, temp, loopCounter, %loop)
                //Init R9 to '0' to start off correctly
                MORPHO {
                        CELL{0,*}:R9 R9=XOR{R9,R9};}
loop:
                stw LUT1, $0, 0xfffff210;
                MORPHO {
                        CELL{0,*} R1 = BYP{FB{mydata, 0x00, OMEGA_BR2, COL_BUS, WORD}};
                        CELL{1,*} NOP{};}
                MORPHO {
                        CELL{0,*} R2 = BYP{FB{mydata, 0x02, OMEGA_BR2, COL_BUS, WORD}};
                        CELL{1,*} NOP{};}
                MORPHO {
                        CELL{0,*} R3 = BYP{FB{mydata, 0x04, OMEGA_BR2, COL_BUS, WORD}};
                        CELL{1,*} NOP{};}
                MORPHO {
                        CELL{0,*} R4 = BYP{FB{mydata, 0x06, OMEGA_BR2, COL_BUS, WORD}};
                        CELL{1,*} NOP{};}
                MORPHO {
```

```
        CELL{*,0}:R8 R8 = PACKH{R0,R1};
        CELL{*,1}:R8 R8 = PACKL{R0,R1};
        CELL{*,2}:R8 R8 = PACKH{R0,R2};
        CELL{*,3}:R8 R8 = PACKL{R0,R2};
        CELL{*,4}:R8 R8 = PACKH{R0,R3};
        CELL{*,5}:R8 R8 = PACKL{R0,R3};
        CELL{*,6}:R8 R8 = PACKH{R0,R4};
        CELL{*,7}:R8 R8 = PACKL{R0,R4};}
// At this point, R8 contains for all columns the low byte.
// There may be a smaller (Omega_array) way to distribute this...
// So, the next step is to look up via a pipeline.

//Prepare the look-up for the tail bytes, a look-up on the register to the left
//To get the result for the first column, we must take the one from the last
// To shift the look-up results by one
//We assume that R9 is initialized to 0 (for the first column) for the first run
MORPHO {
        CELL{*,0}:R8 NOP{};
        CELL{*,1}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,2}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,3}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,4}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,5}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,6}:R8 R9 = BYP{R8{*,$-1}};
        CELL{*,7}:R8 R9 = BYP{R8{*,$-1}};}

CELL{0,*}:R8 NOP{};

//Do the first look-up to get the multiply result
MORPHO {
        CELL{0,*} R1 = BYP{FB{CELL{0,*}, ALL_BANKS, OMEGA_RT, COL_BUS, WORD}};
        CELL{1,*}:R0 NOP{};}

MORPHO {
        CELL{0,*}:R9 NOP{}; //R9 for pipeline
        CELL{1,*}:R0 NOP{};}
//Now second look-up to get the tail bytes
stw LUT2, $0, 0xfffff210;

MORPHO {
        CELL{0,*} R2 = BYP{FB{CELL{0,*}, ALL_BANKS, OMEGA_RT, COL_BUS, WORD}};
        CELL{1,*}:R0 NOP{};}

//XOR the two look-up results to get the final result
MORPHO {
        CELL{0,*}:R7 R7 = XOR{R1,R2};
        CELL{1,*}:R0 NOP{};}

subi loopCounter,loopCounter,1;
addui mydata,mydata,0x8; //Next 8 bytes for input (half a line)

//The convolutional encoded result is in R7. Write this out to memory
//Notice the pipeline.
FB{output,0x0, CELL{0,*}} = CELL{*,*}:OUT_REG NOP{};

// The last trail byte look-up result can go ahead for the next round
MORPHO {
        CELL{*,0}:R8 R9 = BYP{R8{*,$+7}};
        CELL{*,1}:R8 NOP{};
        CELL{*,2}:R8 NOP{};
```

```
                        CELL{*,3}:R8 NOP{};
                        CELL{*,4}:R8 NOP{};
                        CELL{*,5}:R8 NOP{};
                        CELL{*,6}:R8 NOP{};
                        CELL{*,7}:R8 NOP{};}

                // update loop decision
        brlt $0,loopCounter,loop;
addui output,output,0x10; //output moves one line (16 byte)

        MORPHO_ASM_END;
```

3 Optimizing Encoder Performance

The code presented in **Section 2** processes approximately 3.5 input bits per clock cycle, excluding the set-up code. However, the code does not make use of the second row in the RCF array. In a first-level optimization, it should be possible to perform the second look-up from the second row. Also, the read phase of the input, which now requires five MORPHO statements, can be implemented more effectively using the Omega array. We have examined a half-rate encoder for WiMAX, but the mathematics and principles used can be used for other types of encoders. The encoder rate and the constraint length are two factors that determine the design of any encoder.

For example, assume a constraint length of 9 and a one-third rate encoder. One byte at a time, every look-up yields 3 bytes (24 bits) of output data, plus $3 \times 9 = 27$ trail bits. An RCF implementation would require (256×7 *bytes* $\times$ *8 columns* = 14 KB of frame buffer memory, which may too much. Also, more than two look-ups are needed per input byte to store the result in the 16-bit registers. A solution to these issues is to do a nibble-by-nibble look-up. In any case, the principle of a look-up table for implementation is still valid.

Appendix A: WiMAX (802.16D Rev D5) Convolutional Encoder

This appendix presents the information to be placed in the look-up table for a byte-by-byte WiMAX convolutional encoder. The first 16 bits of the look-up results show the output bits; the remaining 12 bits are the trail bits to be XORed with the next look-up result. The ANSII-C code to generate this table is included.

```
0x0000000,0x003bc7,0x00ef1c,0x00d4db,0x003bc70,0x00387b7,0x003536c,0x00368ab,
0x00ef1c0,0x00eca07,0x00e1edc,0x00e251b,0x00d4db0,0x00d7677,0x00da2ac,0x00d996b,
0x03bc700,0x03bfcc7,0x03b281c,0x03b13db,0x0387b70,0x03840b7,0x038946c,0x038afab,
0x03536c0,0x0350d07,0x035d9dc,0x035e21b,0x0368ab0,0x036b177,0x03665ac,0x0365e6b,
0x0ef1c00,0x0ef27c7,0x0eff31c,0x0efc8db,0x0eca070,0x0ec9bb7,0x0ec4f6c,0x0ec74ab,
0x0e1edc0,0x0e1d607,0x0e102dc,0x0e1391b,0x0e251b0,0x0e26a77,0x0e2beac,0x0e2856b,
0x0d4db00,0x0d4e0c7,0x0d4341c,0x0d40fdb,0x0d76770,0x0d75cb7,0x0d7886c,0x0d7b3ab,
0x0da2ac0,0x0da1107,0x0dac5dc,0x0dafe1b,0x0d996b0,0x0d9ad77,0x0d979ac,0x0d9426b,
0x3bc7000,0x3bc4bc7,0x3bc9f1c,0x3bca4db,0x3bfcc70,0x3bff7b7,0x3bf236c,0x3bf18ab,
0x3b281c0,0x3b2ba07,0x3b26edc,0x3b2551b,0x3b13db0,0x3b10677,0x3b1d2ac,0x3b1e96b,
0x387b700,0x3878cc7,0x387581c,0x38763db,0x3840b70,0x38430b7,0x384e46c,0x384dfab,
0x38946c0,0x3897d07,0x389a9dc,0x389921b,0x38afab0,0x38ac177,0x38a15ac,0x38a2e6b,
0x3536c00,0x35357c7,0x353831c,0x353b8db,0x350d070,0x350ebb7,0x3503f6c,0x35004ab,
0x35d9dc0,0x35da607,0x35d72dc,0x35d491b,0x35e21b0,0x35e1a77,0x35eceac,0x35ef56b,
0x368ab00,0x36890c7,0x368441c,0x3687fdb,0x36b1770,0x36b2cb7,0x36bf86c,0x36bc3ab,
0x3665ac0,0x3666107,0x366b5dc,0x3668e1b,0x365e6b0,0x365dd77,0x36509ac,0x365326b,
0xef1c000,0xef1fbc7,0xef12f1c,0xef114db,0xef27c70,0xef247b7,0xef2936c,0xef2a8ab,
0xeff31c0,0xeff0a07,0xeffdedc,0xeffe51b,0xefc8db0,0xefcb677,0xefc62ac,0xefc596b,
0xeca0700,0xeca3cc7,0xecae81c,0xecad3db,0xec9bb70,0xec980b7,0xec9546c,0xec96fab,
0xec4f6c0,0xec4cd07,0xec419dc,0xec4221b,0xec74ab0,0xec77177,0xec7a5ac,0xec79e6b,
```

```
0xe1edc00,0xe1ee7c7,0xe1e331c,0xe1e08db,0xe1d6070,0xe1d5bb7,0xe1d8f6c,0xe1db4ab,
0xe102dc0,0xe101607,0xe10c2dc,0xe10f91b,0xe1391b0,0xe13aa77,0xe137eac,0xe13456b,
0xe251b00,0xe2520c7,0xe25f41c,0xe25cfdb,0xe26a770,0xe269cb7,0xe26486c,0xe2673ab,
0xe2beac0,0xe2bd107,0xe2b05dc,0xe2b3e1b,0xe2856b0,0xe286d77,0xe28b9ac,0xe28826b,
0xd4db000,0xd4d8bc7,0xd4d5f1c,0xd4d64db,0xd4e0c70,0xd4e37b7,0xd4ee36c,0xd4ed8ab,
0xd4341c0,0xd437a07,0xd43aedc,0xd43951b,0xd40fdb0,0xd40c677,0xd4012ac,0xd40296b,
0xd767700,0xd764cc7,0xd76981c,0xd76a3db,0xd75cb70,0xd75f0b7,0xd75246c,0xd751fab,
0xd7886c0,0xd78bd07,0xd7869dc,0xd78521b,0xd7b3ab0,0xd7b0177,0xd7bd5ac,0xd7bee6b,
0xda2ac00,0xda297c7,0xda2431c,0xda278db,0xda11070,0xda12bb7,0xda1ff6c,0xda1c4ab,
0xdac5dc0,0xdac6607,0xdacb2dc,0xdac891b,0xdafe1b0,0xdafda77,0xdaf0eac,0xdaf356b,
0xd996b00,0xd9950c7,0xd99841c,0xd99bfdb,0xd9ad770,0xd9aecb7,0xd9a386c,0xd9a03ab,
0xd979ac0,0xd97a107,0xd9775dc,0xd974e1b,0xd9426b0,0xd941d77,0xd94c9ac,0xd94f26b
```

The source code for table generation is as follows:

```
#include <stddef.h>
#include <stdio.h>

void main (void)
{
        unsigned long i,j,x[16],y[16], k[2];
        unsigned long input=0;
        unsigned long output=0;
        unsigned long status=0;
        unsigned long int inverse[16];
        inverse[0x0]= 0x0;
        inverse[0x1]= 0x8;
        inverse[0x2]= 0x4;
        inverse[0x3]= 0xc;
        inverse[0x4]= 0x2;
        inverse[0x5]= 0xa;
        inverse[0x6]= 0x6;
        inverse[0x7]= 0xe;
        inverse[0x8]= 0x1;
        inverse[0x9]= 0x9;
        inverse[0xa]= 0x5;
        inverse[0xb]= 0xd;
        inverse[0xc]= 0x3;
        inverse[0xd]= 0xb;
        inverse[0xe]= 0x7;
        inverse[0xf]= 0xf;

        for (i=0;i<256;i++)
        {
        k[0]=inverse[i&0x0f];
        k[1]=inverse[(i&0xf0)>>4];
        input=(k[1]<<0)+(k[0]<<4);

       status=input<<6;
       for (j=0;j<15;j++)
       {

x[j]=((status&0x1)>>0)^((status&0x8)>>3)^((status&0x10)>>4)^((status&0x20)>>5)^((status&0x40)>>6);

y[j]=((status&0x1)>>0)^((status&0x2)>>1)^((status&0x8)>>3)^((status&0x10)>>4)^((status&0x40)>>6);
             status=status>>1;
       }

x[15]=x[0]*0x8000000+y[0]*0x4000000+x[1]*0x2000000+y[1]*0x1000000+x[2]*0x800000+y[2]*0x400000+x[3]*0x
200000+y[3]*0x100000+
```

```
        x[4]*0x80000+y[4]*0x40000+x[5]*0x20000+y[5]*0x10000+x[6]*0x8000+y[6]*0x4000+x[7]*0x2000+y[7]*0
               x1000+x[8]*0x800+y[8]*0x400+

x[9]*0x200+y[9]*0x100+x[10]*0x80+y[10]*0x40+x[11]*0x20+y[11]*0x10+x[12]*0x8+y[12]*0x4+x[13]*0x2+y[13]
               *0x1;
                  printf(",0x%x",x[15]);
                }
                return;
```

NOTES:

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu. Minato-ku
Tokyo 106-8573, Japan
0120 191014 or +81-3-3440-3569
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2835
Rev. 0
9/2004