Keywords: CRC, CRC-7, cyclic redundancy check, industrial output, PLC, programmable logic controller, C code, CRC generation, microprocessor, switch

APPLICATION NOTE 6002

CRC PROGRAMMING FOR THE MAX14900E OCTAL, HIGH-SPEED INDUSTRIAL SWITCH

By: Mike van Scherrenburg

Abstract: The MAX14900E octal high-speed industrial switch is a high-performance, feature-rich switch capable of delivering 15W nominal to each of 8 loads. A microprocessor-compatible serial peripheral interface provides access to many advanced features. For added safety, a hardware cyclic redundancy check (CRC) circuit optionally protects this SPI interface against bit errors. This application note provides example C code implementing CRC generation and detection algorithms in the microprocessor.

Introduction

The MAX14900E is a high-performance, 8-channel industrial switch with a rich and advanced feature set. A SPI interface allows a microcontroller to monitor and control most aspects of the MAX14900E. To enhance robustness, a hardware CRC (cyclic redundancy check) circuit in the MAX14900E can optionally protect all data between it and a microprocessor against bit errors.

However, enabling the CRC feature in the MAX14900E is not enough. The microprocessor must also mplement the same CRC algorithm in software, both to append check bits to data being sent to the MAX14900E, as well as to verify those being received from it. One way to accomplish this is to inspect the data sheet, and use it to create custom firmware to implement the necessary CRC functionality.

A better approach would be to use the functions in this application note instead. These are written in C, and should prove easy to port over to most any common microprocessor.

Choosing the Correct Sources

This application note provides six sets of C source code, each implementing a CRC generator and a CRC checker. The MAX14900E communicates with a microprocessor using either single byte packets, or double byte packets. Three of the code sets implement generator and checker algorithms for single byte packets, whereas the other three code sets implement generator and checker algorithms for double byte packets.

Besides the choice of single byte or double byte packets for communication with the MAX14900E, another selection criterion is the importance of code size versus execution speed. This application note provides source code for three different assumptions. First, that small code size is paramount (and that execution speed is secondary). Second, that execution speed is more important than code size. Third, that a balance a desired between code size and execution speed. Curiously, this third “compromise” option proves as fast as the second option in some microprocessor architectures, where table lookups are relatively expensive.

API Conventions Used

Each code set defines two function calls, one for the CRC generator, and another for the CRC checker. Please note that in these code examples, “unsigned char” is an alias for an 8-bit unsigned value, sometimes labeled differently, perhaps something like UINT8. Function return values are either “unsigned char” or “bool”, but some microprocessors execute code faster if the return values are “int” instead, the native storage size of that microprocessor.

The single byte generator function call looks something like:

send2 = generator (send1);

Send1 is the byte to send to the MAX14900E. The code should transmit send1 followed by send2 to the MAX14900E. Likewise, the double byte generator function call looks something like this:
send3 = generator (send1, send2);

The user should send 3 bytes, send1, followed by send2, then send3, over the SPI interface.

While the microprocessor sends configuration settings to the MAX14900E over the SPI interface, the MAX14900E simultaneously returns status information back to the microprocessor. The single byte checker looks something like:

flag = checker (ret1, ret2);

The Boolean “flag” is true (!= 0) if the return byte and CRC match. The double byte checker looks like:

flag = checker (ret1, ret2, ret3);

Small Code Size

The code examples in this section use loops to build the CRC one bit at a time. For this reason, they are not so fast, but compile compactly. These code snippets rely on a helper routine, Loop_CRC, to perform the actual bit-at-a-time CRC calculation.

```
unsigned char Loop_CRC (unsigned char crc, unsigned char byte)
{
        int i;

        for (i = 0; i < 8; i++)
        {
                crc <<= 1;
                if (crc & 0x80)
```

```
                        crc ^= 0xB7; // 0x37 with MSBit on purpose
                if (byte & 0x80)
                        crc ^=1;
                byte <<= 1;
        }

        return crc;
}

//
unsigned char crcSmallEncode8 (unsigned char byte1)
{
        unsigned char synd;

        synd = Loop_CRC (0x7f, byte1);
        return Loop_CRC (synd, 0x80) | 0x80;
}

//
bool crcSmallCheck8 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;

        synd = Loop_CRC (0x7f, byte1);
        return Loop_CRC (synd, byte2) == 0;
}



unsigned char Loop_CRC (unsigned char crc, unsigned char byte)
{
        int i;

        for (i = 0; i < 8; i++)
        {
                crc <<= 1;
                if (crc & 0x80)
                        crc ^= 0xB7; // 0x37 with MSBit on purpose
                if (byte & 0x80)
                        crc ^=1;
                byte <<= 1;
        }
```

```
        return crc;
}

//
unsigned char crcSmallEncode16 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;

        synd = Loop_CRC (0x7f, byte1);
        synd = Loop_CRC (synd, byte2);
        return Loop_CRC (synd, 0x80) | 0x80;
}

//
bool crcSmallCheck16 (unsigned char byte1, unsigned char byte2, unsigned char
 byte3)
{
        unsigned char synd;

        synd = Loop_CRC (0x7f, byte1);
        synd = Loop_CRC (synd, byte2);
        return Loop_CRC (synd, byte3) == 0;
}
```

Fast Execution

The code examples in this section trade code size for speed. Two 128-byte tables encode one full byte of data per table access. Note that if the user only wishes to protect data sent to the MAX14900E, and uses single byte transactions, then the user only needs the second table.

```
//
const unsigned char propagate8_next0 [128] = {
 0x00, 0x6E, 0x6B, 0x05, 0x61, 0x0F, 0x0A, 0x64,
 0x75, 0x1B, 0x1E, 0x70, 0x14, 0x7A, 0x7F, 0x11,
 0x5D, 0x33, 0x36, 0x58, 0x3C, 0x52, 0x57, 0x39,
 0x28, 0x46, 0x43, 0x2D, 0x49, 0x27, 0x22, 0x4C,
 0x0D, 0x63, 0x66, 0x08, 0x6C, 0x02, 0x07, 0x69,
 0x78, 0x16, 0x13, 0x7D, 0x19, 0x77, 0x72, 0x1C,
 0x50, 0x3E, 0x3B, 0x55, 0x31, 0x5F, 0x5A, 0x34,
 0x25, 0x4B, 0x4E, 0x20, 0x44, 0x2A, 0x2F, 0x41,
 0x1A, 0x74, 0x71, 0x1F, 0x7B, 0x15, 0x10, 0x7E,
 0x6F, 0x01, 0x04, 0x6A, 0x0E, 0x60, 0x65, 0x0B,
```

```
 0x47, 0x29, 0x2C, 0x42, 0x26, 0x48, 0x4D, 0x23,
 0x32, 0x5C, 0x59, 0x37, 0x53, 0x3D, 0x38, 0x56,
 0x17, 0x79, 0x7C, 0x12, 0x76, 0x18, 0x1D, 0x73,
 0x62, 0x0C, 0x09, 0x67, 0x03, 0x6D, 0x68, 0x06,
 0x4A, 0x24, 0x21, 0x4F, 0x2B, 0x45, 0x40, 0x2E,
 0x3F, 0x51, 0x54, 0x3A, 0x5E, 0x30, 0x35, 0x5B
};

const unsigned char propagate8_next1 [128] = {
 0xB7, 0xD9, 0xDC, 0xB2, 0xD6, 0xB8, 0xBD, 0xD3,
 0xC2, 0xAC, 0xA9, 0xC7, 0xA3, 0xCD, 0xC8, 0xA6,
 0xEA, 0x84, 0x81, 0xEF, 0x8B, 0xE5, 0xE0, 0x8E,
 0x9F, 0xF1, 0xF4, 0x9A, 0xFE, 0x90, 0x95, 0xFB,
 0xBA, 0xD4, 0xD1, 0xBF, 0xDB, 0xB5, 0xB0, 0xDE,
 0xCF, 0xA1, 0xA4, 0xCA, 0xAE, 0xC0, 0xC5, 0xAB,
 0xE7, 0x89, 0x8C, 0xE2, 0x86, 0xE8, 0xED, 0x83,
 0x92, 0xFC, 0xF9, 0x97, 0xF3, 0x9D, 0x98, 0xF6,
 0xAD, 0xC3, 0xC6, 0xA8, 0xCC, 0xA2, 0xA7, 0xC9,
 0xD8, 0xB6, 0xB3, 0xDD, 0xB9, 0xD7, 0xD2, 0xBC,
 0xF0, 0x9E, 0x9B, 0xF5, 0x91, 0xFF, 0xFA, 0x94,
 0x85, 0xEB, 0xEE, 0x80, 0xE4, 0x8A, 0x8F, 0xE1,
 0xA0, 0xCE, 0xCB, 0xA5, 0xC1, 0xAF, 0xAA, 0xC4,
 0xD5, 0xBB, 0xBE, 0xD0, 0xB4, 0xDA, 0xDF, 0xB1,
 0xFD, 0x93, 0x96, 0xF8, 0x9C, 0xF2, 0xF7, 0x99,
 0x88, 0xE6, 0xE3, 0x8D, 0xE9, 0x87, 0x82, 0xEC
};

//
unsigned char crcFastEncode8 (unsigned char byte1)
{
        unsigned char synd;

        synd = (byte1 & 0x80) ? 0xEC : 0x5B; // 6C & 5B before optimization
        synd ^= byte1;
        return propagate8_next1[synd];
}

bool crcFastCheck8 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;
        unsigned char const *ptr;
```

```
        synd = (byte1 & 0x80) ? 0xEC : 0x5B;
        synd ^= byte1;
        ptr = byte2 & 0x80 ? propagate8_next1 : propagate8_next0;
        return (ptr[synd] ^ byte2) == 0;
}

//
const unsigned char propagate8_next0 [128] = {
 0x00, 0x6E, 0x6B, 0x05, 0x61, 0x0F, 0x0A, 0x64,
 0x75, 0x1B, 0x1E, 0x70, 0x14, 0x7A, 0x7F, 0x11,
 0x5D, 0x33, 0x36, 0x58, 0x3C, 0x52, 0x57, 0x39,
 0x28, 0x46, 0x43, 0x2D, 0x49, 0x27, 0x22, 0x4C,
 0x0D, 0x63, 0x66, 0x08, 0x6C, 0x02, 0x07, 0x69,
 0x78, 0x16, 0x13, 0x7D, 0x19, 0x77, 0x72, 0x1C,
 0x50, 0x3E, 0x3B, 0x55, 0x31, 0x5F, 0x5A, 0x34,
 0x25, 0x4B, 0x4E, 0x20, 0x44, 0x2A, 0x2F, 0x41,
 0x1A, 0x74, 0x71, 0x1F, 0x7B, 0x15, 0x10, 0x7E,
 0x6F, 0x01, 0x04, 0x6A, 0x0E, 0x60, 0x65, 0x0B,
 0x47, 0x29, 0x2C, 0x42, 0x26, 0x48, 0x4D, 0x23,
 0x32, 0x5C, 0x59, 0x37, 0x53, 0x3D, 0x38, 0x56,
 0x17, 0x79, 0x7C, 0x12, 0x76, 0x18, 0x1D, 0x73,
 0x62, 0x0C, 0x09, 0x67, 0x03, 0x6D, 0x68, 0x06,
 0x4A, 0x24, 0x21, 0x4F, 0x2B, 0x45, 0x40, 0x2E,
 0x3F, 0x51, 0x54, 0x3A, 0x5E, 0x30, 0x35, 0x5B
};

const unsigned char propagate8_next1 [128] = {
 0xB7, 0xD9, 0xDC, 0xB2, 0xD6, 0xB8, 0xBD, 0xD3,
 0xC2, 0xAC, 0xA9, 0xC7, 0xA3, 0xCD, 0xC8, 0xA6,
 0xEA, 0x84, 0x81, 0xEF, 0x8B, 0xE5, 0xE0, 0x8E,
 0x9F, 0xF1, 0xF4, 0x9A, 0xFE, 0x90, 0x95, 0xFB,
 0xBA, 0xD4, 0xD1, 0xBF, 0xDB, 0xB5, 0xB0, 0xDE,
 0xCF, 0xA1, 0xA4, 0xCA, 0xAE, 0xC0, 0xC5, 0xAB,
 0xE7, 0x89, 0x8C, 0xE2, 0x86, 0xE8, 0xED, 0x83,
 0x92, 0xFC, 0xF9, 0x97, 0xF3, 0x9D, 0x98, 0xF6,
 0xAD, 0xC3, 0xC6, 0xA8, 0xCC, 0xA2, 0xA7, 0xC9,
 0xD8, 0xB6, 0xB3, 0xDD, 0xB9, 0xD7, 0xD2, 0xBC,
 0xF0, 0x9E, 0x9B, 0xF5, 0x91, 0xFF, 0xFA, 0x94,
 0x85, 0xEB, 0xEE, 0x80, 0xE4, 0x8A, 0x8F, 0xE1,
 0xA0, 0xCE, 0xCB, 0xA5, 0xC1, 0xAF, 0xAA, 0xC4,
 0xD5, 0xBB, 0xBE, 0xD0, 0xB4, 0xDA, 0xDF, 0xB1,
 0xFD, 0x93, 0x96, 0xF8, 0x9C, 0xF2, 0xF7, 0x99,
```

```
 0x88, 0xE6, 0xE3, 0x8D, 0xE9, 0x87, 0x82, 0xEC
};

//
// This is the fast (one byte at a time) algorithm
//
// This is so fast that one could benefit from running it in-line
//
unsigned char crcFastEncode16 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;
        unsigned char const *ptr;

        synd = (byte1 & 0x80) ? 0xEC : 0x5B;
        synd ^= byte1;
        ptr = byte2 & 0x80 ? propagate8_next1 : propagate8_next0;
        synd = ptr[synd] ^ byte2;
        return propagate8_next1[synd];
}


bool crcFastCheck16 (unsigned char byte1, unsigned char byte2, unsigned char
byte3)
{
        unsigned char synd;
        unsigned char const *ptr;

        synd = (byte1 & 0x80) ? 0xEC : 0x5B;
        synd ^= byte1;
        ptr = byte2 & 0x80 ? propagate8_next1 : propagate8_next0;
        synd = ptr[synd] ^ byte2;
        ptr = byte3 & 0x80 ? propagate8_next1 : propagate8_next0;
        return (ptr[synd] ^ byte3) == 0;
}
```

Balanced

The code examples in this section strike a balance between code size and execution speed. One table lookup encodes 7 bits of data into the CRC. The eighth bit is encoded using techniques used in the size-optimized code. Some Harvard architecture devices have relatively expensive table lookup execution times, and for this class of microprocessor, this balanced code might execute as quickly as the speed optimized code, or perhaps even faster.

```
//
unsigned char propagate7 [128] = {
 0x00, 0x6E, 0xDC, 0xB2, 0xD6, 0xB8, 0x0A, 0x64,
 0xC2, 0xAC, 0x1E, 0x70, 0x14, 0x7A, 0xC8, 0xA6,
 0xEA, 0x84, 0x36, 0x58, 0x3C, 0x52, 0xE0, 0x8E,
 0x28, 0x46, 0xF4, 0x9A, 0xFE, 0x90, 0x22, 0x4C,
 0xBA, 0xD4, 0x66, 0x08, 0x6C, 0x02, 0xB0, 0xDE,
 0x78, 0x16, 0xA4, 0xCA, 0xAE, 0xC0, 0x72, 0x1C,
 0x50, 0x3E, 0x8C, 0xE2, 0x86, 0xE8, 0x5A, 0x34,
 0x92, 0xFC, 0x4E, 0x20, 0x44, 0x2A, 0x98, 0xF6,
 0x1A, 0x74, 0xC6, 0xA8, 0xCC, 0xA2, 0x10, 0x7E,
 0xD8, 0xB6, 0x04, 0x6A, 0x0E, 0x60, 0xD2, 0xBC,
 0xF0, 0x9E, 0x2C, 0x42, 0x26, 0x48, 0xFA, 0x94,
 0x32, 0x5C, 0xEE, 0x80, 0xE4, 0x8A, 0x38, 0x56,
 0xA0, 0xCE, 0x7C, 0x12, 0x76, 0x18, 0xAA, 0xC4,
 0x62, 0x0C, 0xBE, 0xD0, 0xB4, 0xDA, 0x68, 0x06,
 0x4A, 0x24, 0x96, 0xF8, 0x9C, 0xF2, 0x40, 0x2E,
 0x88, 0xE6, 0x54, 0x3A, 0x5E, 0x30, 0x82, 0xEC
};

//}
unsigned char crcMediumEncode8 (unsigned char byte1)
{
        unsigned char synd;

        synd = (byte1 ^ 0xEC);
        if (synd & 0x80)
                synd ^= 0xB7;
        synd = propagate7[synd] ^ 0x80;
        if (synd & 0x80)
                synd ^= 0xB7;
        return synd | 0x80;
}

//
bool crcMediumCheck8 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;

        synd = (byte1 ^ 0xEC);
        if (synd & 0x80)
                synd ^= 0xB7;
```

```
        synd = propagate7[synd] ^ byte2;
        if (synd & 0x80)
                synd ^= 0xB7;
        return synd == 0;
}

//
unsigned char propagate7 [128] = {
 0x00, 0x6E, 0xDC, 0xB2, 0xD6, 0xB8, 0x0A, 0x64,
 0xC2, 0xAC, 0x1E, 0x70, 0x14, 0x7A, 0xC8, 0xA6,
 0xEA, 0x84, 0x36, 0x58, 0x3C, 0x52, 0xE0, 0x8E,
 0x28, 0x46, 0xF4, 0x9A, 0xFE, 0x90, 0x22, 0x4C,
 0xBA, 0xD4, 0x66, 0x08, 0x6C, 0x02, 0xB0, 0xDE,
 0x78, 0x16, 0xA4, 0xCA, 0xAE, 0xC0, 0x72, 0x1C,
 0x50, 0x3E, 0x8C, 0xE2, 0x86, 0xE8, 0x5A, 0x34,
 0x92, 0xFC, 0x4E, 0x20, 0x44, 0x2A, 0x98, 0xF6,
 0x1A, 0x74, 0xC6, 0xA8, 0xCC, 0xA2, 0x10, 0x7E,
 0xD8, 0xB6, 0x04, 0x6A, 0x0E, 0x60, 0xD2, 0xBC,
 0xF0, 0x9E, 0x2C, 0x42, 0x26, 0x48, 0xFA, 0x94,
 0x32, 0x5C, 0xEE, 0x80, 0xE4, 0x8A, 0x38, 0x56,
 0xA0, 0xCE, 0x7C, 0x12, 0x76, 0x18, 0xAA, 0xC4,
 0x62, 0x0C, 0xBE, 0xD0, 0xB4, 0xDA, 0x68, 0x06,
 0x4A, 0x24, 0x96, 0xF8, 0x9C, 0xF2, 0x40, 0x2E,
 0x88, 0xE6, 0x54, 0x3A, 0x5E, 0x30, 0x82, 0xEC
};

//
unsigned char crcMediumEncode16 (unsigned char byte1, unsigned char byte2)
{
        unsigned char synd;

        synd = (byte1 ^ 0xEC);
        if (synd & 0x80)
                synd ^= 0xB7;
        synd = propagate7[synd] ^ byte2;
        if (synd & 0x80)
                synd ^= 0xB7;
        synd = propagate7[synd] ^ 0x80;
        if (synd & 0x80)
                synd ^= 0xB7;
        return synd | 0x80;
}
```

```
//
bool crcMediumCheck16 (unsigned char byte1, unsigned char byte2, unsigned cha
r byte3)
{
        unsigned char synd;

        synd = (byte1 ^ 0xEC);
        if (synd & 0x80)
                synd ^= 0xB7;
        synd = propagate7[synd] ^ byte2;
        if (synd & 0x80)
                synd ^= 0xB7;
        synd = propagate7[synd] ^ byte3;
        if (synd & 0x80)
                synd ^= 0xB7;
        return synd == 0;
}
```

Comparisons

To get some idea how efficient these CRC calculations are, this code was implemented on a Microchip PIC24F family running at 16 MIPS (32MHz internal CPU clock). This is a 16-bit data path microprocessor with 24-bit instructions. The only modifications to the code were to change the return values to “short”, which the C compiler interprets as a signed 16-bit value, its native data width.

Code size is determined from compiler and linker tables. An ancillary program generates 1,024 random 8-, 16-, or 24-bit values. Times reported are an average of these 1,024 tests. **Table 1** summarizes the data obtained using the Microchip C compiler and their code emulator. For code using table lookups, the size is specified in bytes as CONST.

Table 1. Execution Time and Code Size Comparison Using a PIC24

Algorithm	Bytes (PGM + CONST)	Encode Time (µs)	Decode Time (µs)
Compact Single Byte	219	24.8	25.5
Speedy Single Byte	570 (186 + 384)	1.72	3.00
Balanced Single Byte	323 (195 + 128)	2.06	2.44
Compact Double Byte	267	37.2	37.8
Speedy Double Byte	723 (339 + 384)	3.19	4.34
Balanced Double Byte	473 (345 + 128)	3.02	3.40

Different microprocessors will yield different results, but in the case of the PIC24F, the speedy algorithms need not be used. If interested in a fast CRC conversion, both the speedy and the balanced algorithms should be benchmarked on the target processor to determine which is better.

Conclusion

This application note has shown how to code CRC algorithms on a microprocessor communicating with a MAX14900E octal high-speed industrial switch. Taking advantage of the C code examples in this application note is a worry-free way to implement this extra protection.

In some cases, some benchmarking should be performed on the target processor, especially if fast execution speed is a priority.

Related Parts		
MAX14900E	Octal, High-Speed, Industrial, High-Side Switch	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/en/support
For Samples: http://www.maximintegrated.com/en/samples
Other Questions and Comments: http://www.maximintegrated.com/en/contact

Application Note 6002: http://www.maximintegrated.com/en/an6002
APPLICATION NOTE 6002, AN6002, AN 6002, APP6002, Appnote6002, Appnote 6002