Carnegie Mellon University
Research Showcase @ CMU

Institute for Software Research | School of Computer Science

2006

Efficient High Hamming Distance CRCs for Embedded Networks

Justin Ray
Carnegie Mellon University

Philip Koopman
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/isr

This Conference Proceeding is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Institute for Software Research by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Preprint, Dependable Systems and Networks (DSN), June 25-28 2006, Philadelphia PA.

Efficient High Hamming Distance CRCs for Embedded Networks

Justin Ray, Philip Koopman
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15217
Email: {justinr2, koopman}@ece.cmu.edu

Abstract

Cyclic redundancy codes (CRCs) are widely used in network transmission and data storage applications because they provide better error detection than lighter weight checksum techniques. 24- and 32-bit CRC computations are becoming necessary to provide sufficient error detection capability (Hamming distance) for critical embedded network applications. However, the computational cost of such CRCs can be too high for resource-constrained embedded systems, which are predominantly equipped with 8-bit microcontrollers that have limited computing power and small memory size. We evaluate the options for speeding up CRC computations on 8-bit processors, including comparing variants of table lookup approaches for memory cost and speed. We also evaluate classes of CRC generator polynomials which have the same computational cost as 24- or 16-bit CRCs, but provide 32-bit CRC levels of error detection, and recommend good polynomials within those classes for data word lengths typical of embedded networking applications.

1 Introduction

Using cyclic redundancy codes (CRCs) for error detection in embedded systems involves a tradeoff among speed, memory consumption, and error detection effectiveness. Because many embedded systems have significant resource constraints, it is important to understand the available tradeoff options and, if possible, find ways to attain better error detection at lower computational cost. In this paper we analyze existing algorithm tradeoffs to quantify typical tradeoff parameters for embedded applications. Additionally, we identify two new classes of 32-bit CRCs that can be calculated with the same computational cost as existing 16- and 24-bit CRCs while providing improved error detection effectiveness. For these special case polynomials, we have computed the bound for error detection and provided a list of polynomials with good error detection performance.

In general, codes that provide better error detection require greater effort to compute. The primary drawback to the CRC is its computational cost, which is much higher than simpler error codes such as the Fletcher checksum or other addition-based checksums [7]. For the embedded domain, computational cost can be a major design factor because of the severe cost constraints on many systems. However, for those applications that must attain high levels of error detection, CRCs are the only practical alternative proven in field use.

CRCs are commonly used in enterprise, desktop, and high-end embedded applications, including standards such as Ethernet [12], ATM networks [4], and IEEE 1394(Firewire™) [10]. More recently, CRCs with high Hamming distances have become increasingly important for deeply embedded systems. The Hamming distance (HD) of an error code is the minimum number of bit errors that must be present to potentially be undetected. For example, a Hamming distance 6 code (HD=6) guarantees detection of up to 5 bit errors in a single network message, but fails to detect some fraction of possible 6-bit errors.

Safety critical embedded applications in particular require high Hamming distances. Applications such as automotive X-by-Wire protocols [8, 26] and train control networks typically require HD=6 at all message lengths. High HD CRCs are also employed as auxiliary protection mechanisms, often called "safety CRCs," to provide additional error detection beyond the capability of ordinary network protocols. For example, the Multifunction Vehicle Bus (MVB) train network uses an 8-bit CRC for each 64-bit packet of data transmitted at the link layer. But the MVB logical frame format (which can be as long as 256 bytes) uses a 32-bit CRC, called a "safety code" [16] to provide HD=6 protection for critical messages. Another rail example of this is the "vital CRC" given in [11].

A particularly demanding constraint is that embedded networks usually have a mix of high-end and low-end nodes, and even the lowest cost node on a system must be able to compute CRC values quickly enough to keep up with

network traffic. This requirement becomes even more difficult if high-level services such as a protocol's group membership approach require the active participation of all network nodes. Because of this, we focus this paper on understanding CRCs performance tradeoffs for 8-bit microcontrollers, because they are by far the most prevalent devices being used in embedded systems [27].

While CRCs have been in use for decades, it is difficult to find engineering guidance for them, and even harder to find comprehensive design tradeoff information. In order to explore the CRC computation in the embedded domain, we have implemented various known CRC algorithms in modern 8-bit processors and have analyzed performance and resource requirements. In the process of implementing these algorithms, we have identified some discrepancies in the existing literature, including confusion about what constitutes a "correct" software implementation, incorrect check values, and problems with data processing order.

In addition to studying existing algorithms, we have developed and evaluated techniques which are optimized for a special class of 32-bit CRC checksums that speed up calculations and reduce memory requirements while achieving good error detection performance. We also present the result of an exhaustive search of the space of these special polynomials wherein we define the Hamming distance bound and identify a list of "good" polynomials.

The remainder of this paper will focus on analyzing the algorithms for and performance of correct, efficient implementations of the CRC algorithm in the embedded domain. Section 2 discusses the background and related work in this area. Section 3 explores the tradeoffs among various algorithms when implementation in low-end processors. Section 4 describes a novel class of CRC generator polynomials and how they may be used to achieve better performance and error detection. Section 5 describes the experiments we performed to measure the tradeoffs of various implementations. Section 6 compares the performance of new and existing algorithms. Section 7 describes the correct implementation of the core CRC computation and identifies known implementation issues. Section 8 summarizes the paper and provides recommendations for system designers.

2 Background and Related Work

CRCs are widely used for error detection in a variety of applications. Despite their prevalence, there are significant gaps in understanding the engineering tradeoffs in their use. Commonly, there are even larger gaps between known best approaches and common engineering practices.

2.1 Terminology

Some terms used in the following discussion are:

data word — the data that is fed into the CRC computation to produce the checksum.

Table 1. Bitwise Left-Shift CRC Algorithm

```
for (i=0; i<sizeof(data); i++) {
  if (msb(data) ^ msb(crc)) {
    crc = (crc << 1) ^ (poly);
  } else {
    crc = (crc << 1);
  }
  data <<= 1;
}
```

frame check sequence (FCS) — the value produced by the CRC computation. This digest or checksum provides the redundant information necessary for error detection.

code word — the data word with the FCS appended

undetected error — result of an error which happens to corrupt bits in the code word in such a way that it produces another valid code word. It is important to note that corruptions can and often do occur in both the data word and FCS portions of a code word.

burst error — an error pattern stated in terms of a length m (i.e. an m-bit burst error) where two up to m bit errors may occur exclusively in an m bit range.

Hamming distance (HD) — in the context of error detection, the minimum number of bits in the code word that must be independently corrupted in order to cause an undetected error. For a CRC, the HD depends on the data word length, the FCS length, and the generator polynomial used. For example, the polynomial $x^8 + x^5 + x^2 + x^1 + x^0$, which has HD=4 for data words of 18 to 55 bits, will detect all 1-, 2-, and 3-bit errors for those lengths.

2.2 Mathematical Foundation

Mathematically, the CRC algorithm used to generate the FCS can be described as modulo-two polynomial division. Binary data can be represented as a polynomial where the bit values are the coefficients of various powers of x. In other words, the data byte "01001001" can be represented as "$0 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 1 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$". The CRC checksum is defined by the equation:

$$crc(x) = \left(data(x) \cdot x^k\right) mod\, g(x)$$

where $g(x)$ is the "generator polynomial" of order k. A more detailed description can be found in [24].

A C implementation of the CRC is given in Table 1. For clarity, we will refer to this as the "left-shift algorithm." This algorithm processes the data most significant bit (MSB) first, so the data is *left-shifted* through the register, hence the name.

A polynomial of a given order m has $m + 1$ terms (from x^m term to the x^0 term). In the software implementation, the generator polynomial is an m-bit binary representation of the CRC polynomial, where the most significant term (x^m) is not actually present, but is implicitly understood to be present. For example, the CCITT-16 polynomial (given

in [24] as $x^{16} + x^{12} + x^5 + 1$) should be represented as $0x1021$. Polynomials discussed in [14] have a binary expression which includes most significant term (x^m) but has an implicit lowest order term (x^0). Either form can be used, depending on the implementation, but when specifying polynomials, it is important to specify which form is being used. The polynomial representation is unambiguous.

While arithmetic checksum codes can only provide HD=2 or HD=3 for most data lengths, CRCs can give much higher HD for the same FCS length. In addition, they detect all burst errors up to the length of the FCS, and greater than 99% of burst errors longer than the FCS length [2]. As previously mentioned, critical embedded applications usually have a high HD requirement of HD=6.

There are five basic parameters that affect the FCS output by the CRC implementations in real systems: 1) CRC polynomial, 2) initial CRC value, 3) final value which is XORed in the CRC register, 4) order in which data bits are processed, and 5) order in which CRC bits are placed into the FCS field.

A partial list of these parameters for various standards can be found in [1, 28]. A change in any of these parameters will affect the final FCS value. The orders of bit processing and in which bits are placed into the FCS must be consistent to preserve burst error properties (this is discussed further in a Section 7). Having a non-zero initial value can be useful for detecting bit-slip errors in data with a series of leading 0 bits. The initial and final values do not affect HD; only the CRC polynomial and data word length affect HD.

Given a maximum message length and required HD performance, the art of selecting a CRC is choosing a good generator polynomial, $g(x)$, which determines the error detecting performance of the CRC checksum [14, 5]. Conventional wisdom suggests picking one of the "standard" polynomials is safer than choosing one at random (e.g., [1]). However, many "standard" polynomials have poor performance, or suboptimal performance compared to identified "good" polynomials [14]. As we will discuss later, there are some special case situations where other polynomials may be chosen to achieve more efficient implementation.

In the discussion that follows, we use the notation *CRC32*, *CRC24*, and *CRC16* to refer to a generic (i.e. no specific generator polynomial) CRC implementations which have a 32-, 24- and 16-bit FCS sizes, respectively. Any reference to a particular standard, such as the *CRC32 standard*, will be named specifically.

2.3 Related Work

There are a number of algorithms for producing an FCS with desirable error detection properties, including arithmetic checksums (e.g. Fletcher checksum), weighted sum codes (WSC), and cyclic redundancy codes (CRC). Arithmetic checksums are employed in TCP because they can be computed very efficiently [3], but safety-critical and high-reliability systems typically require error codes with HD of six or greater [26, 8]. CRCs can achieve this, but at a higher computational cost [18] than other checksum approaches.

Error detecting codes in general and cyclic redundancy codes in particular have been studied for many years. Castagnoli *et al.* [5], Lin *et al.* [17], and Peterson *et al.* [23] are standard references in the field. However, until recently the difficult problem of finding optimal codes remained unsolved because of the significant amount of computing power required to examine all possible codes. Recent advances enabled exhaustive searches for optimal polynomials for CRC3 up to CRC16 to identify the optimal HD bound [14]. [13] also presents the results for exhaustive search for HD=6 polynomials for data words lengths up to and beyond the Ethernet MTU size.

There is a significant amount of research into improving CRC performance by using optimized or parallel implementations in special purpose hardware (e.g. VLSI [21] and FPGA [20]). While this approach is essential for some high-speed applications, there is still a need for software-based implementations to support the common situation where off-the-shelf components are used to reduce costs and improve time to market.

Software implementations of the CRC take various forms and have been published in [1, 6, 22, 24, 25, 28]. They detail various algorithms which we will discuss in Section 3. Research addressing the relative speed of various algorithms has been presented by [7, 15]. Generally, performance studies deal with the effects of memory caching and number of instructions for various algorithms on high-end processors with large memory caches. However, deeply embedded system designers need to understand performance on processors that usually have no cache memory and have different instruction sets.

We are aware of occasional instances where specialized CRC polynomials such as the ones we discuss to speed up computations have been considered for use in industry projects. However, we have not been able to find any discussion of this topic in the academic literature, nor any published analysis of performance tradeoffs or discussion of which polynomials perform well in such situations.

3 CRC Implementations for Embedded Processors

A key tradeoff in implementing CRC error detection is the memory space versus computation time tradeoff of the algorithm used to compute the checksum. In this section we examine different algorithms that use varying amounts of memory to speed up CRC computations beyond the simple, but slow, left-shift algorithm already described.

8-bit processors are sensitive to this tradeoff because they often have relatively slow clocks and limited instruc-

tion sets. They also have restricted memory capacity and narrow addressing buses. For an embedded protocol to be viable in many situations, it must be feasible to implement the protocol on these processors using some type of optimized CRC algorithm. In our examination of performance tradeoffs, we focus on code word lengths of up to 2048 bits, which is at or well beyond maximum message lengths for typical embedded networks.

For these implementations, we examine optimized assembly implementations in two 8-bit microcontroller architectures, the PIC16F series from Microchip™[19] and the HS08 series from Freescale™[9]. These were selected because they are representative of typical small microcontrollers in widespread use. The PIC16F has a Harvard architecture, meaning it has separate buses for program and data memory. All instructions take 4 clock cycles to execute, except jump instructions, which take 8 clock cycles (due to prefetch queue flushing). The HS08 architecture is a von Neumann machine, with a single memory space for addressing data RAM and program memory. Instructions take a varying number of clock cycles to complete. Both architectures utilize an accumulator register for arithmetic operations. Every attempt was made to introduce an equivalent implementation in both systems, but some differences remain due to the inherently different natures of the architectures. These differences are of minimal importance because our goal is simply to illustrate typical approaches.

We wish to emphasize that our goal is not to analyze the relative performance of the Microchip and Freescale products, to evaluate their fitness for the purpose of computing CRCs, or to recommend their use for embedded or safety-critical embedded applications. Rather, our goal is to show that there is a performance relationship among the various algorithms, and that the trends in performance are general rather than peculiar to only a single architecture.

In the remainder of this section, we describe four typical classes of implementation for the CRC algorithm, and compare the performance tradeoffs in low-end embedded systems. For each CPU architecture, assembly language implementations were developed for each software algorithm for CRC16, CRC24, CRC32, and the special purpose optimized polynomials. Implementations of CRC8 were not considered because that size CRC cannot achieve high enough HD for critical applications.

3.1 Bit-shift Algorithm

The bit-shift algorithm (BSA) is exactly the algorithm previously described in Section 2 and Figure 1. This "basic" algorithm is directly derived from the binary division operation. It is simple to implement and requires minimal program memory. However, the loop must execute once for every bit of the data word, hence it is also the slowest executing algorithm.

3.2 Table Lookup Algorithm

The table lookup algorithm (TLA) is described in [22] and [25]. This algorithm can update the accumulated FCS value for multiple bits of the data word in a single computation. The TLA is optimized through the use of a precomputed table of values. Each table entry is the CRC checksum of the table index value, and is k bits in length (the block size of the FCS). Processing n bits of data at a time requires table of size $k * 2^n$ bits. Because the entries depend only on the CRC generator polynomial, they can be computed and stored at design time. When the algorithm is being run, the index for the table lookup is a combination of the current CRC value and the new data. As the algorithm iterates over each n bits of data, the current CRC value is shifted by n bits and XORed with the table entry.

For our implementation, the algorithm iterates the computation over 8-bit blocks of data ($n = 8$), with table entries being the same width as the FCS. Thus the total table sizes are 512, 768, and 1024 bytes for CRC16, CRC24, and CRC32, respectively. Because the 8-bit architectures only have an 8-bit data bus, the table is organized as several 256-byte tables, each requiring a separate computed goto or memory fetch (depending on the architecture). The TLA is commonly considered the fastest executing algorithm, but its memory footprint can be prohibitively large for embedded processors that might only have a few kilobits of memory. In a real system, the high memory requirement is likely to be further exacerbated by memory paging problems. In both architectures, the table lookup instructions use an 8-bit operand, so each 256-byte table must start on a page boundary. Any memory between the end of the program instructions and the beginning of the table entries is therefore wasted.

3.3 Virtual Table Algorithm

The virtual table algorithm (VTA) is based on an algorithm in [24]. Like the TLA, it can operate on multiple bits of the data word concurrently. However, instead of retrieving the table entry from memory, a virtual table entry is computed on the fly. The table entries are computed based on the contribution of each bit of index value to the final table value, with each virtual table entry computation resulting in the identical value that would have been fetched from memory had a physical table been precomputed and stored in memory per the TLA approach.

For processing n bits of data at a time, the index can be denoted by the XOR of single bit values $b_0 - b_{n-1}$:

$$index = (b_{n-1} \times (2^{n-1})) \oplus \ldots \oplus (b_1 \times (2^1)) \oplus b_0$$

Then because the CRC is linear over XOR, we have:

$$\begin{aligned} crc(index) = \quad & crc(b_{n-1} \times (2^{n-1})) \\ & \oplus \ldots \\ & \oplus crc(b_1 \times (2^1)) \\ & \oplus crc(b_0) \end{aligned}$$

Thus, for processing n bits of data at a time, only the precomputed values $crc(b_i \times 2^i), i \epsilon \{0, \ldots, n\}$ are stored. The index is computed and the current CRC value shifted (as in the TLA). If bit i of the index is a 1, the value $crc(b_i \times 2^i)$ is XORed with the CRC.

This algorithm is similar to the "reduced table algorithm" described in [24]. We prefer to term this algorithm as "virtual table" rather than "reduced table" because an important aspect of this algorithm is that the values of $crc(b_i \times 2^i)$ do not depend on the current state of the computation. Thus the precomputed values can be hard coded into the routine and do not actually require the overhead associated with multiple table lookups (this optimization is not mentioned in [24]).

In general, the VTA is faster than the BSA (because of bytewise processing), but slower than the TLA. However, the memory savings over TLA are substantial because there is very little memory spent on a table – n in-line data entries rather than a 2^n entry table plus the code to fetch the values.

3.4 Optimized Virtual Table Algorithm

The optimized virtual table algorithm (OVTA) is a special case of the VTA, and is suggested by [6], as well as being similar to the on-the-fly algorithm suggested by [22]. Rather than computing table entries by bit-testing various positions in the index, the table entry can in some cases be constructed by observing patterns in the table values and devising a series of shift-and-XOR operations.

As we will show, the OVTA can be faster than the VTA. In some cases, it can even approach the speed of the TLA. There is one important difference between this algorithm and the others. For the BSA, TLA, and VTA, the computational speed of the algorithm is independent of the polynomial chosen. However, for the OVTA, the optimization over VTA depends completely on the characteristics of the generator polynomial chosen. Table 2 shows the improvement over the VTA for several different polynomials (refer to Section 4 for a description of CRC32sub8 and CRC32sub16) . Note that for the particular CRC24 and CRC32 polynomials we used for our experiments, the OVTA has no improvement at all over the VTA. This is because the pattern of bits was too complicated to develop a shift-and-XOR implementation that was faster than the VTA. The performance of the VTA represents the upper bound on the performance of OVTA, because a designer can fall back to the VTA if the optimization strategy has a negative effect on performance.

4 Algorithm Optimization for Special Polynomial Classes

The four algorithmic approaches discussed in Section 3 provide speed vs. memory size tradeoff points widely used in current CRC implementations. However, it is possible to get further speed increases for some of the methods by careful selection of CRC polynomials.

Table 2. Speedup in Worst-Case Execution Time for Optimized Virtual Table Algorithm

Algorithm	Polynomial	Architecture	
		PIC	HS08
CRC16	1021	42.1%	55.0%
CRC32sub8	0x000001ED	11.5%	30.2%
CRC24*	0x5D6DCB	0.0%	0.0%
CRC32sub16	0x0001B435	29.9%	34.1%
CRC32*	0x4C11DB7	0.0%	0.0%

** 0% speedup reflects an algorithm where no polynomial specific optimization could be made over standard virtual table algorithm.*

It seems obvious that the CRC32 computation should require more cycles than the CRC24 computation, which in turn should require more cycles than the CRC16 computation. In this section, we examine the actual source of the additional overhead and identify some optimizations that can be used to improve the speed of the computation of CRC32 to rival CRC24 and CRC16 computation speed while simultaneously improving error detection capabilities to CRC32 levels of error detection (for shorter data words).

For each algorithm described in Section 3, all arithmetic and shift operations must be done byte-by-byte. For example, an XOR in the generic algorithm (Table 1) requires 2, 3, and 4 XOR operations in the embedded implementations of the CRC16, CRC24, and CRC32, respectively. By eliminating some of these operations, the performance of the CRC algorithm can be significantly improved.

We begin this optimization by observing the characteristics of certain CRC32 polynomials. We specify this class of polynomials as CRC$k$sub$r$, with the form[1] :

$$x^k + a_r * x^r + \ldots + a_1 * x^1 + a_0 * x^0$$

These polynomials have the desirable characteristic that the generator polynomial and the resulting table entries have whole bytes of zeros. Table 3 compares table values for a regular CRC32 polynomial to a CRC32sub8 polynomial. Note that the upper two bytes of the CRC32sub8 polynomial table entries are always 0.

Because XOR with a zero value is the identity operation, using the CRC$k$sub$r$ polynomials allows us to eliminate some of the overhead of the CRC computation of longer FCS values. In the BSA, we only need to consider the non-zero bytes when XORing in the CRC polynomial. In the TLA, VTA, and OVTA, the size of the table entries (and therefore the computational cost of looking them up or com-

[1] The reverse polynomials (i.e. those of the form $x^k + a_{k-1} * x^{k-1} + \ldots + a_{k-r} * x^{k-r} + a_0 * x^0$) have the similar table properties and identical error performance, so we omit them from this discussion.

Table 3. Sample Table-Lookup Entries for CRC32 and CRC32sub8

Table Index	CRC32sub8 (0x000001ed)	CRC32 (0x04c11db7)
0	0x00000000	0x00000000
1	0x000001ed	0x04c11db7
...	...	...
254	0x0000a5b6	0xb5365d03
255	0x0000a45b	0xb1f740b4

Note that the upper two bytes of the table entries for the polynomial 0x000001ED are always 0.

puting them) is also reduced. Additionally, this approach reduces the lookup table memory size for the TLA algorithm.

The CRC32sub8 table has two non-zero bytes in it (one for the low 8 bits of result, and one to account for left-shift propagation of up to 8 bits for byte-by-byte processing). Intuitively, this makes the computational cost similar to a normal CRC16 computation, which also has two-byte tables. Similarly, the CRC32sub16, which has three non-zero bytes, has performance similar to that of the CRC24 computation. The CRC$k$sub$r$ polynomials are slightly slower because there is some additional overhead to handling the larger FCS that cannot be eliminated.

It is important to note that the increased computational speed of the CRC$k$sub$r$ polynomials is not without some tradeoffs. The size of the FCS is still k bits and cannot be reduced. Therefore, increased bandwidth or storage size for the larger FCS (and correspondingly larger code word) is an additional cost of this approach. However, at times when the error detection effectiveness of a 32-bit CRC is desired at reduced cost, this technique can prove useful.

5 Experiments

In order to implement the various CRC algorithms, we obtained development tools from Microchip (for the PIC16F) and Freescale (for the HS08). We implemented each of the four algorithms (BSA, TLA, VTA, and OVTA) for each class of CRC polynomials (CRC16, CRC24, CRC32, CRC32sub8, and CRC32sub16) on both architectures. All code development was done in assembly, and the resulting programs were simulated using cycle-accurate software tools provided by the respective manufacturers. These tools allowed us to measure execution times and memory requirements for each algorithm exactly.

In our experiment, we measure worst-case execution time (WCET), average execution time (AET), and best-case execution time (BCET). WCET is the longest possible path through the code, and BCET is the shortest path. These measurements were taken using simulation tools to force the code into longer or shorter branch paths. AET was determined by measuring the execution time required to compute the CRC over 512 byte samples of random data. Some algorithms (notably the TLA) have a fixed execution sequence, so the WCET, AET, and BCET are equal.

For memory requirements, we measure the total number of program memory words required to implement the algorithm, including memory for table entries and any memory required for storage of program code. The algorithms with the heaviest memory usage are the TLA implementations. Although program memory words are 14 bits in the PIC16F and 8 bits in the HS08, a 256 byte table requires 256 program memory words in either case. The additional memory required by the PIC implementation is not "wasted" because the PIC architecture does not allow program memory to be read directly. Because we did not set out to compare the performance of these particular processors (or architectures, for that matter), we do not consider these differences to be germane, so we simply report results in terms of total memory words used.

6 Results

We now describe the results of the experiments described in Section 5. We demonstrate the cost of the CRC algorithms in terms of execution time and memory requirements. We also analyze the relative error detection capabilities of the various algorithms. Because we have introduced new classes of CRC polynomials (CRC32sub8 and CRC32sub16), we also present a list of "good" polynomials in those classes for various Hamming Distances and code word lengths.

6.1 Performance

Figures 1 and 2 compare the computational speed of the four algorithms for CRC16, CRC32sub8, CRC24, CRC32sub16, and CRC32 on each microcontroller. WCET, AET, and BCET are represented by the narrowing bars. Bars that do not narrow represent implementations with fixed execution paths. Figures 3 and 4 compare the memory requirements for the same set of algorithms. As expected, there is clearly a tradeoff between memory usage and execution speed: faster execution can be obtained at the expense of increased memory usage.

As expected, the performance of CRC16 and CRC32sub8 are roughly comparable, as are CRC24 and CRC32sub16.

6.2 Error Detection Capability

When choosing an algorithm and generator polynomial for performance, it is important not to overlook the error detection capability of various design choices. To compare error detection capabilities, we computed the HD bound for CRC32sub8, CRC24, and CRC32sub16 according to the methods described in [14] (see Section 6.3 for a short de-

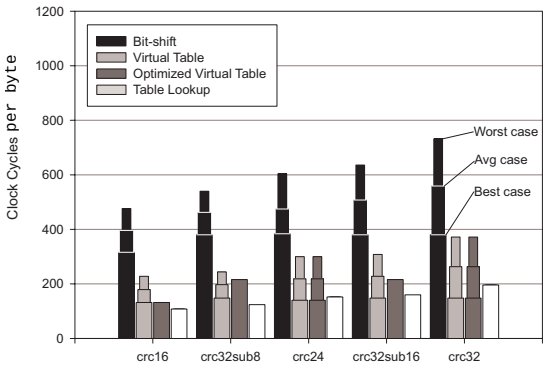

Figure 1. Execution Time for PIC16F

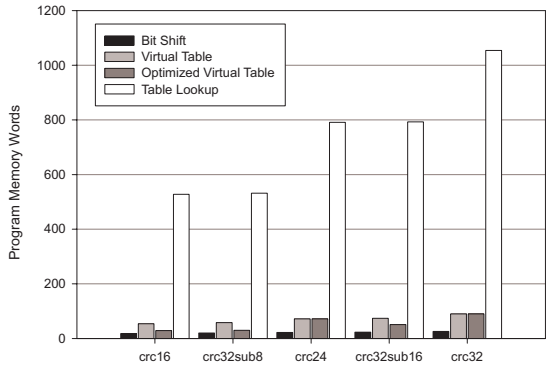

Figure 3. Memory Use for PIC16F

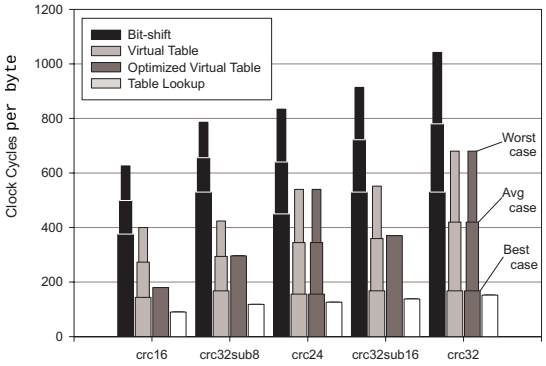

Figure 2. Execution Time for HS08

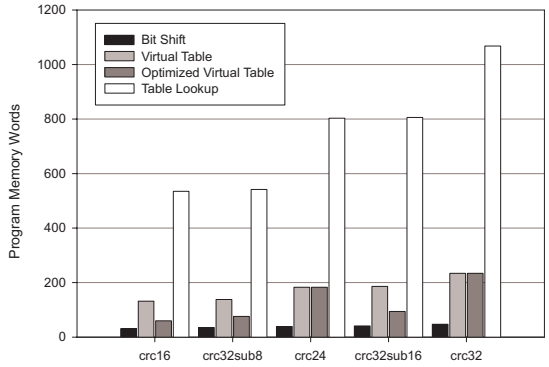

Figure 4. Memory Use for HS08

scription). The results are shown in Table 4. For each class of polynomials, the table shows the maximum code word length for which any polynomial of that class can achieve the stated HD. For example, there is a CRC16 polynomial which can provide HD=8 for code words from 36 to 151 bits in length; beyond 151 bits, only HD=5 or less is possible. The bound for CRC32 is not shown because it is not known; the computation of this bound is currently considered intractable. However, 32-bit polynomials are known that can provide HD=6 out to almost 32K bits [5], so as a practical matter the tradeoff of using other polynomials is the maximum length at which HD=6 can be provided.

While CRC16 and CRC32sub8 have roughly equivalent computation and memory cost, CRC32sub8 actually provides significantly better error detection. For 2048 bit code words (a reasonable maximum size for embedded network messages), CRC32sub8 provides HD=6, while the best CRC16 polynomial provides only HD=4, an improvement of two additional bits of HD. Because all CRCs provide burst error detection up to the length of the FCS regardless of polynomial, the CRC32sub8 polynomials also provide superior burst error detection.

When comparing CRC24 and CRC32sub16, which also have similar memory and performance costs, it is clear that CRC32sub16 has superior error performance. In addition, CRC32sub8, which is *faster to compute* than CRC24 also has better error detection properties for code words longer than 151 bits. As before, CRC32sub8 and CRC32sub16 both provide burst error detection for all bursts up to 32 bits in width. Because a chosen CRC must provide a given HD for the entire range of possible messages, the choice of either CRC32sub8 or CRC32sub16 for appropriately sized code words would be an improvement in error detection over CRC24 with the same or better computational speed. Another benefit of adopting CRC32sub8 or CRC32sub16 algorithms to replace CRC24 is future extensibility. 2048 bits is the *maximum* code word length at which CRC24 can provide HD=6, which is required for most safety-critical applications, so any future expansion would require a CRC algorithm with HD=6 coverage beyond this length. CRC32sub8 can provide HD=6 for code words of up to 4145 bits (more than double that of CRC24), and CRC32sub16 up to 8220 bits (more than 4 times that of CRC24).

6.3 Good CRC$k$sub$r$ Polynomials

The optimal HD bound and optimum polynomials were obtained according to the method used in [14], which we describe briefly here. For each HD, the entire space of polynomials meeting the CRC$k$sub$r$ criteria was evaluated to the longest data length where a polynomial of the given HD still exists. If there is only one polynomial which reaches

Table 4. Hamming Distance and Maximum Code Word Length

	Hamming Distance (bits)						
	12	11	10	9	8	7	6
CRC16	–	–	–	–	31	35	**151**
CRC32sub8	–	–	–	–	197	270	**4145**
CRC24	47	–	50	63	129	255	**2048**
CRC32sub16	62	65	106	116	313	516	**8046+**

This table shows the maximum code word length (data word + FCS) for which a given HD can be obtained for a given polynomial class.

this length, then it is considered good. If there are multiple such polynomials, then the one with the lowest Hamming weight (number of undetected errors at a given HD) is identified. All polynomials within 10% of this minimum are then searched to further identify which ones achieve a still higher HD for shorter message lengths, and the one which achieves that higher HD for the longest message length is considered the best. Other criteria are possible for selecting good polynomials, but this set of criteria provides a reasonable engineering tradeoff for use on typical embedded applications.

Table 5 shows a breakdown of the good polynomials for the CRC32sub8 polynomial class. Table 6 lists the good polynomials for CRC32sub16 polynomial class. Each polynomial is given as a numeric value (binary representation with implicit x^{32} term), as well as a polynomial representation. The third line of each entry contains a list of the degrees of the polynomial's factors, using the notation from [13]. Each good polynomial provides the stated HD (or better) for all code words less than or equal to the stated length. Although the bound for HD=6 is at code words of 4145 bits, a good polynomial for code words with a maximum length of 2048 bits is also given, because it has better error detection at short data lengths.

7 Correctness of the CRC Algorithm

One of the practical issues in implementing CRCs is ensuring the correct bit order of computation and placement of bits into the FCS to preserve burst error properties. While most network protocols do this properly, it is a tricky area that is not always implemented correctly.

Suppose that an application implements a standard which requires the data bits to be processed least significant bit (LSB) first, as in the CRC32 standard. Because the bit-reversal process is slow in most processors, instead of using a "left-shift" algorithm, implementers might instead develop an equivalent "right-shift" algorithm, which is shown in Table 7. This algorithm uses the *reversed* CRC polynomial with implicit x^0 term. The reverse of the CCITT-16 polynomial is $x^{16} + x^{11} + x^4 + 1$ and should be represented

Table 5. Good CRC32sub8 Polynomials

HD	Polynomial	Length
8	0x000001D7 $x^{32}+x^8+x^7+x^6+x^4+x^2+x^1+x^0$ $\{1, 1, 5, 25\}$	197
7	0x00000179 $x^{32}+x^8+x^6+x^5+x^4+x^3+x^0$ $\{2, 30\}$	270
6	0x000001ED $x^{32}+x^8+x^7+x^6+x^5+x^3+x^2+x^0$ $\{1, 10, 21\}$	2048
6	0x000000E5 $x^{32}+x^7+x^6+x^5+x^2+x^0$ $\{1, 1, 3, 4, 23\}$	4145

This table shows good polynomials and the maximum code word length obtained for the stated Hamming Distance. Each entry gives the polynomial in the "left-shift" binary notation and the standard polynomial notation. The third line of each entry list the orders of the polynomials prime factorization.

as $0x8408$. It is important to note that not only does this algorithm process data bits LSB first, it also results in a bit-reversed CRC value (compared to the result if the bits were fed into a left-shift algorithm MSB first).

The left-shift versus right-shift issue is one source of confusion in the literature. While [28, 1] describe the left-shift algorithm, [6] describes the right-shift algorithm as though it were equivalent and lists incorrect check values in the paper.

Conventional wisdom suggests that either algorithm can be used as long as the correct assumptions about data ordering are made [6, 28]. This misconception probably stems from the well-known property of CRCs that a polynomial and its bit-reversed counterpart have identical error detection performance.

But the left-versus-right shift issue is not so simple for real implementations. The problem arises because software implementations of the CRC algorithm rely on data registers of limited width and limited lookup table size. Table 8 shows a concrete C implementation of the left-shift algorithm. In essence, the core algorithm processes data one byte at a time. While the example code could easily be expanded to process 16- or 32-bit words as well, the issue is that in many cases the FCS size is larger than the chunk size in which the data is processed. The execution of the outer loop creates artificial boundaries in the data. Changing the core algorithm to a right-shift implementation would swap the order of the bits *within the registers* without affecting the overall order that the data words are processed in.

As long as the code that generates the checksum processes data bits in the same order as the code that verifies the checksum, the checksum value would be verified correctly, and bit-ordering would not be an impediment to

Table 6. Good CRC32sub16 Polynomials

HD	Polynomial	Length
12	0x0001DA97 $x^{32}+x^{16}+x^{15}+x^{14}+x^{12}+x^{11}+$ $x^{9}+x^{7}+x^{4}+x^{2}+x^{1}+x^{0}$ $\{1,2,11,18\}$	62
11	0x00015A67 $x^{32}+x^{16}+x^{14}+x^{12}+x^{11}+x^{9}+x^{6}+x^{5}+x^{2}+x^{1}+x^{0}$ $\{3,5,8,16\}$	65
10	0x00018AD5 $x^{32}+x^{16}+x^{15}+x^{11}+x^{9}+x^{7}+x^{6}+x^{4}+x^{2}+x^{0}$ $\{1,1,11,19\}$	106
9	0x00008D35 $x^{32}+x^{15}+x^{11}+x^{10}+x^{8}+x^{5}+x^{4}+x^{2}+x^{0}$ $\{32\}$	116
8	0x0000B3E1 $x^{32}+x^{15}+x^{13}+x^{12}+x^{9}+x^{8}+x^{7}+x^{6}+x^{5}+x^{0}$ $\{1,8,9,14\}$	313
7	0x00002979 $x^{32}+x^{13}+x^{11}+x^{8}+x^{6}+x^{5}+x^{4}+x^{3}+x^{0}$ $\{5,9,9,9\}$	516
6	0x00003551 $x^{32}+x^{13}+x^{12}+x^{10}+x^{8}+x^{6}+x^{4}+x^{0}$ $\{1,5,13,13\}$	8220+

This table is arranged in the same way as Table 5.

superficial testing of system operation. However, for serial transmission, it is important that the data bits be processed *in the same order that they are sent down the wire*. Otherwise, a k bit (where k is the length of the FCS) burst error straddling the artificial boundaries created by the software algorithm can affect bits that are actually further than k bits apart from the point of view of the CRC algorithm. Figure 5 illustrates this concept.

An example of this problem can be found in the IEEE1394 specification [10], which uses the CRC32 standard polynomial to protect both the header and data segments of the packet. Packet contents are divided into 32-bit chunks called quadlets. The sample algorithm given in the specification is a quadlet-at-a-time algorithm which processes bits MSB first, but the quadlets are transmitted LSB first.

This bit ordering problem only affects the burst error detection property of the CRC. The data protection for random independent bit errors is not affected. It may not be feasible to modify existing standards, especially for a standard like IEEE1394, which is not generally relied upon for safety-critical applications. But it is important to understand this problem so that new standards can take full advantage of the error-detecting capabilities of the CRC.

Table 7. Bitwise Right-Shift CRC Algorithm

```
for (i=0; i<sizeof(data); i++) {
  if (lsb(data) ^ lsb(crc)) {
    crc = (crc >> 1) ^ (revpoly);
  } else {
    crc = (crc >> 1);
  }
  data >>= 1;
}
```

Table 8. C Code for 8-bit Left-Shift Algorithm

```
int datalen;
unsigned char crc = crc_init;
unsigned char data[datalen];
int i, j;
unsigned char gpoly = 0xEA;
for (i=0; i<datalen; i++) {
  /* begin core algorithm */
  for (j=0; j<8; j++) {
    if (lsb(data) ^ lsb(crc)) {
      crc = (crc << 1) ^ (gpoly_brev);
    } else {
      crc = (crc << 1);
    }
    data[i] <<= 1;
  }
  /* end core algorithm */
}
```

8 Conclusion

Cyclic redundancy codes are commonly used to provide error detection for network messages and stored data because they can provide better minimum Hamming distances than other checksums. In particular, high HD codes are increasing in importance in safety critical embedded system applications. We have identified several algorithms and studied their implementation in low-end embedded processors to identify tradeoffs among error detection, computation speed, and memory requirements. We have also documented the concept of the CRC$k$sub$r$ polynomial and illustrated how implementations taking advantage of the special characteristics of these polynomials can achieve better error detection, more efficient computational speeds, and smaller memory requirements.

We summarize existing algorithms for computing the CRC, including the bit-shift algorithm, table lookup algorithm, virtual table algorithm, and optimized virtual table algorithm. By implementing these algorithms in the several embedded architectures, we are able to compare the performance and memory tradeoffs. Additionally, we have implemented algorithms that are optimized for two novel classes of polynomials: CRC32sub8 and CRC32sub16. Computing CRCs with these polynomials offers improved error de-

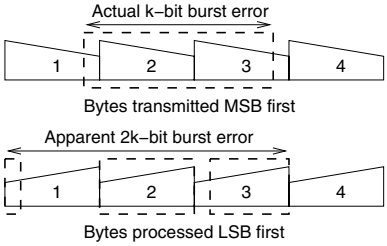

Figure 5. Undetectable burst error caused by byte inversion

tection with computational performance that is comparable to the performance of algorithms for smaller polynomial classes (e.g. CRC16, CRC24). We also present polynomial selection tables for these new classes of polynomials.

Because the mathematical basis of the CRC is not necessarily intuitive to engineers used to dealing with ordinary integer arithmetic, it can be difficult to obtain a correct implementation of the software algorithm. We have clarified some discrepancies in existing literature and identified a real application in which incorrect specification of bit ordering compromises CRC error detection capability.

Overall, we hope that these results provide embedded application engineers with better tradeoff information for selecting CRC algorithms and polynomials to attain good tradeoffs among speed, memory consumption, and error detection effectiveness.

Acknowledgment

The authors would like to thank Bombardier Transportation and Honeywell for their generous support.

References

[1] M. Barr. Slow and steady never lost the race. *Embedded Systems Programming*, pages 37–46, January 2000.

[2] P. E. Boudreau, W. C. Bergman, and D. R. Irvin. Performance of a cyclic redundancy check and its interaction with a data scrambler. *IBM Journal of Research Development*, 38(6):651–658, 1994.

[3] R. Braden, D. Borman, and C. Partridge. RFC1071: Computing the internet checksum. Online: http://www.faqs.org/-rfcs/rfc1071.html, 1988.

[4] F. Braun and M. Waldvogel. Fast incremental CRC updates for IP over ATM networks. In *2001 IEEE Workshop on High Performance Switching and Routing*, pages 48–52, 2001.

[5] G. Castagnoli, S. Bräuer, and M. Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Trans. Comm.*, 41(6):883–892, 1993.

[6] J. Crenshaw. Implementing CRCs. *Embedded Systems Programming*, January 1992.

[7] D. C. Feldmeier. Fast software implementation of error detection codes. *IEEE/ACM Trans. Netw.*, 3(6):640–651, December 1995.

[8] FlexRay-Consortium. FlexRay communications system, protocol specification, version 2.0. Request online: http://-www.flexray.com/specification_request.php.

[9] Freescale Semiconductor, Inc. HCS08 microcontrollers. Online: http://www.freescale.com/files/microcontrollers/-doc/data_sheet/MC9S08RG60.pdf, 2003.

[10] IEEE Std 1394-1995. IEEE standard for a high performance serial bus, Aug 1996. ISBN 0-7381-1203-8.

[11] IEEE Std 1570-2002. IEEE standard for the interface between the rail subsystem and the highway subsystem at a highway rail intersection, 2002. ISBN 0-7381-3397-1.

[12] IEEE Std 802.3-2000. Part 3: Carrier sense multiple access with collisition detection (CSMA/CD) access method and physical layer specification, 2000. ISBN: 0-7381-2673-X.

[13] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *International Conference on Dependable Systems and Networks*, pages 459–498, 2002.

[14] P. Koopman. Cyclic redundancy code (CRC) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks*, pages 145–154, 2004.

[15] M. E. Kounavis and F. L. Berry. A systematic approach to building high performance software-based CRC generators. In *Proceedings of the 10th IEEE Symposium on Computers and Communications*, 2005.

[16] G. Krut. Justification for the format of safety telegram. AD-tranz corporation technical document, 1996.

[17] Lin, Shu, and D. Costello. *Error Control Coding*. Prentice-Hall, 1983.

[18] A. J. McAuley. Weighted sum codes for error detection and their comparison with existing codes. *IEEE/ACM Trans. Netw.*, 2(1):16–22, 1994.

[19] Microchip Technology, Inc. PIC16F7X data sheet. Online: http://www.microchip.com/downloads/en/DeviceDoc/-41206a.pdf, 2002.

[20] F. Monteiro, A. Dandache, A. M'sir, and B. Lepley. A fast CRC implementation on FPGA using a pipelined architecture for the polynomial division. In *the 8th IEEE International Conference on Electronics Circuits and Systems*, volume 3, pages 1231 – 1234, 2001.

[21] T.-B. Pei and C. Zukowski. High-speed parallel CRC circuits in VLSI. *IEEE Transactions on Communications*, 40(4):653–657, 1992.

[22] A. Perez. Byte-wise CRC calculations. *IEEE Micro*, 3(3):40–50, 1983.

[23] W. Peterson and E. Weldon. *Error-Correcting Codes*. MIT Press, second edition, 1972.

[24] T. V. Ramabadran and S. S. Gaitonde. A tutorial on CRC computations. *IEEE Micro*, 8(4):62–75, 1988.

[25] D. V. Sarwate. Computation of cyclic redundancy checks via table look-up. *Commun. ACM*, 31(8):1008–1013, 1988.

[26] TTA-Group. Time-triggered protocol TTP/C, high-level specification document, protocol version 1.1. Request online: http://www.ttagroup.org/technology/specification.htm, 2003.

[27] J. Turley. Embedded processors, part one. Online: http://www.extremetech.com/print_article/-0,3998,a=21014,00.asp, 2002.

[28] R. Williams. A painless guide to CRC error detection. Online: http://www.ross.net/crc/download/crc_v3.txt, 1993.