DOT/FAA/TC-14/49

Federal Aviation Administration
William J. Hughes Technical Center
Aviation Research Division
Atlantic City International Airport
New Jersey 08405

Selection of Cyclic Redundancy Code and Checksum Algorithms to Ensure Critical Data Integrity

March 2015

Final Report

U.S. Department of Transportation
Federal Aviation Administration

Technical Report Documentation Page

1. Report No. DOT/FAA/TC-14/49	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subitle SELECTION OF CYCLIC REDUNDANCY CODE AND CHECKSUM ALGORITHMS TO ENSURE CRITICAL DATA INTEGRITY	5. Report Date March 2015 6. Performing Organization Code 220410
7. Author(s) Philip Koopman[1], Kevin Driscoll[2], Brendan Hall[2]	8. Performing Organization Report No.
9. Performing Organization Name and Address [1]Carnegie Mellon University 5000 Forbes Ave., Pittsburgh, PA 15213 USA [2]Honeywell Laboratories 1985 Douglas Drive North, Golden Valley, MN 55422 USA	10. Work Unit No. (TRAIS) 11. Contract or Grant No. DTFACT-11-C-00005
12. Sponsoring Agency Name and Address U.S. Department of Transportation Federal Aviation Administration 950 L'Enfant Plaza SW, 5th Floor Washington, DC 20024	13. Type of Report and Period Covered Final Report, August 2011-April 2013 14. Sponsoring Agency Code\ AIR-134

15. Supplementary Notes

The Federal Aviation Administration Aviation Research Division COR was Charles Kilgore.

16. Abstract

This report explores the characteristics of checksums and cyclic redundancy codes (CRCs) in an aviation context. It includes a literature review, a discussion of error detection performance metrics, a comparison of various checksum and CRC approaches, and a proposed methodology for mapping CRC and checksum design parameters to aviation integrity requirements. Specific examples studied are Institute of Electrical and Electronics Engineers (IEEE) 802.3 CRC-32; Aeronautical Radio, Incorporated (ARINC)-629 error detection; ARINC-825 Controller Area Network (CAN) error detection; Fletcher checksum; and the Aeronautical Telecommunication Network (ATN)-32 checksum. Also considered are multiple error codes used together, specific effects relevant to communication networks, memory storage, and transferring data from nonvolatile to volatile memory.

Key findings include: (1) significant differences exist in effectiveness between error-code approaches, with CRCs being generally superior to checksums in a wide variety of contexts; (2) common practices and published standards may provide suboptimal (or sometimes even incorrect) information, requiring diligence in selecting practices to adopt in new standards and new systems; (3) error detection effectiveness depends on many factors, with the Hamming distance of the error code being of primary importance in many practical situations; (4) no one-size-fits-all error-coding approach exists, although this report does propose a procedure that can be followed to make a methodical decision as to which coding approach to adopt; and (5) a number of secondary considerations must be taken into account that can substantially influence the achieved error-detection effectiveness of a particular error-coding approach.

17. Key Words CRC, Cyclic redundancy code, Checksum, Error coding, Error detection, Network data errors, Memory data errors	18. Distribution Statement This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 111	22. Price

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

ACKNOWLEDGEMENTS

The authors would like to thank the following people for their contributions to this project and the underlying work upon which it was based:

- Justin Ray, Carnegie Mellon University
- Theresa Maxino, Carnegie Mellon University
- Tridib Chakravarty, Carnegie Mellon University
- Pavan Allalaghatta, Honeywell

The authors would like to acknowledge the following Federal Aviation Administration Review Team individuals for providing support to the project:

- Liz Brandli
- Michael P. DeWalt
- Gary Horan
- Charles Kilgore
- Barbara Lingberg
- Srini Mandalapu
- Steve Paasch
- John Strasburger
- Will Struck

The authors would also like to thank the anonymous contributors who provided valuable information during our industry data collection.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

AC	Advisory Circular
ARINC	Aeronautical Radio, Incorporated
ATN	Aeronautical Telecommunication Network
BER	Bit error ratio
CAN	Controller Area Network
CCITT	Comité Consultatif International Téléphonique et Télégraphique
COTS	Commercial, Off-the-Shelf
CRC	Cyclic redundancy code
FAA	Federal Aviation Administration
FCS	Frame check sequence
HD	Hamming distance
HDLC	High Level Data Link Control
HW	Hamming weight
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet protocol
Kbytes	Kilobytes (1024 bytes—since random-access memory capacity, such as CPU cache measurements are always stated in multiples of 1024 (2^{10}) bytes, due to memory's binary addressing)
LFSR	Linear feedback shift register
LRC	Longitudinal redundancy check
MIC	Message Integrity Check
Modbus	Modicon bus
NRZ	Non-return-to-zero
NUREG	Nuclear Regulatory Commission publication
PROFIBUS	Process field bus
PROFIsafe	Process field bus safety profile
Pud	Probability of undetected error
RAM	Random access memory
RFC	Request for comment
RTCA	RTCA, Inc. (formerly Radio Technical Commission for Aeronautics)
RZ	Return-to-zero
SAE	SAE, Inc.
SCTP	Stream Control Transmission Protocol
SIL	Safety Integrity Level
TCP	Transmission Control Protocol
TTP/C	Time-Triggered Protocol Version C
UDP	User Datagram Protocol
XOR	Exclusive OR

EXECUTIVE SUMMARY

This report explores the characteristics of checksums and cyclic redundancy codes (CRCs) in an aviation context. It includes a literature review; a discussion of error detection performance metrics; a comparison of various checksum and CRC approaches; and a proposed methodology for mapping CRC and checksum design parameters to aviation integrity requirements. Specific examples studied are Institute of Electrical and Electronics Engineers (IEEE) Standard 802.3 CRC-32; Aeronautical Radio, Incorporated (ARINC)-629 error detection; ARINC-825 Controller Area Network error detection; Fletcher checksum; and the Aeronautical Telecommunication Network-32 checksum. Also considered are multiple error codes used together, specific effects relevant to communication networks, memory storage, and transferring data from nonvolatile to volatile memory.

The key findings of the report show that:

- Significant differences in performance between error detection approaches, with CRCs, in many cases, providing dramatically better error-detection capabilities than checksums.

- Common practice and published standards have suboptimal[1] (or sometimes incorrect) information about error detection codes. For example, Fletcher checksums are sometimes said to be as good as a standard CRC, but are often dramatically less effective than a well-chosen CRC. This report provides state-of-the-art information in this area.

- Error detection effectiveness depends on multiple factors, including the size of data to be protected, the types of data errors expected, the smallest number of bit errors that are undetectable by the error code (Hamming distance [HD]), and the undetected error fraction of errors as a function of number of bit errors present in a particular piece of data.

- There is no one-size-fits-all answer as to which error detection code to use. In evaluating error detection capability, a recommended high-level approach is to:

  – Understand the acceptable probability of undetected error (Pud)

  – Characterize exposure to data faults

  – Determine what fraction of dangerous faults will be undetected by a particular error code, given an appropriate fault model

  – Determine if the Pud (based on both exposure to faults and fault detection rate) is sufficiently low to meet integrity requirements

[1] The use of the term "suboptimal," when describing a technique, should not be interpreted as indicating that the technique is not suitable for particular purposes.

In determining if undetected error probabilities are low enough, it can be sufficient to pick an error code with an adequately large HD so that all errors expected to occur within a system's operating life are guaranteed to be detected (so long as a random independent fault model accurately reflects the errors that will be seen in operation).

There are a number of other considerations, constraints, and system-level issues that must be considered. These include:

- The need to scrub error detection mechanisms and data values to mitigate the risk of fault accumulation over time

- Vulnerabilities due to message framing (e.g., corrupted length field undermining CRC effectiveness)

- Vulnerabilities due to bit encoding (e.g., stuff bits, multi-bit symbols, or scrambling undermining CRC HD)

- Potential bit error correlations due to memory geometry

- Complex intermediate processing stages potentially invalidating an assumption of unpatterned, random independent bit errors

1. INTRODUCTION.

1.1 PURPOSE.

This document is the final report of a task to create a comparative summary of checksum and Cyclic Redundancy Code (CRC) performance in an aviation context. Specific goals are to perform a literature review of evaluation criteria, parameters, and tradeoffs; study CRC and checksum performance with respect to their design parameters and achieved error detection performance; recommend a way to relate CRC and checksum design parameters to functional integrity levels; and make recommendations for future research work.

This report also briefly addresses additional areas vital to correct application of error detection codes in practice. These areas include the effects of data encoding formats on error detection code effectiveness and the error detection effectiveness of using multiple error codes. The scope of the report is limited to checksums and CRCs.

1.2 BACKGROUND.

While checksums and CRCs have been in use for many years (the seminal CRC reference dates back to 1957 [1]), published research into their effectiveness is sparse. Additionally, many different application domains, including aviation, use a wide variety of data-integrity approaches that do not necessarily correspond to published best practices. In part, this seems to be because of computational costs that have historically limited the ability to evaluate CRC performance. Another factor is a communication gap between mathematical discussions of the techniques and what practitioners need to successfully apply those techniques to aviation and other domains. Often, it is simply because of practitioners copying problematic error detection techniques already in use under the incorrect assumption that widely used approaches must be good approaches. The research covered by this report is intended to address these issues.

The general approach of this research was to complete the following steps:

1. Perform a literature review to identify relevant error detection coding techniques and candidate evaluation criteria for checksums and CRCs.

2. Consider representative aviation applications available to the authors, including data characteristics, usage scenarios, and fault exposure.

3. Evaluate error detection code effectiveness via simulation and analysis.

4. Suggest a strategy for mapping error-detection effectiveness to functional integrity levels.

5. Identify topics for future work.

A tutorial on checksums and CRCs was created to provide technical background and is included in presentation slide format as appendix C of this report.

1.3 RELATED ACTIVITIES, DOCUMENTS, SYMBOLOGY, AND TERMINOLOGY.

The Statement of Work for this project specifically identifies Koopman and Chakravarty [2] as the basis for the literature review. A literature review with a description of more detailed publications in this area can be found in section 7.

The Glossary section of this report provides working definitions for terminology.

To minimize complex typography, scientific notation using calculator-style "e" notation has been used. For example, "2.36e-6" is 2.36 times 10 raised to the -6 power, equivalent to the number 0.00000236. For CRC polynomials, the exponentiation symbol "^" has been used in keeping with the most common convention for that type of mathematics. The CRC polynomials are not evaluated to produce numeric answers in this report, whereas the scientific notation numbers and the other uses of exponentiation are numerically evaluated.

In this report, the terms "fault" and "error" are used interchangeably with regard to data corruption, which is common in this area of study, but does not necessarily encompass the distinctions made between those terms in system-level dependability studies.

2. METHODOLOGY AND VALIDITY.

2.1 FAULT MODEL AND UNDETECTED ERRORS.

The principal purpose of this work is to understand how often undetected errors can occur and how to relate that information to integrity levels. The fault model is a binary symmetric channel with bit inversion faults (sometimes called a "bit flip" fault model). In other words, a data value is represented in bits and each bit error results in a single bit value being inverted from 0 to 1 or from 1 to 0.

An undetected error is considered to have occurred when a corrupted codeword is valid. A codeword is the concatenation of a particular dataword and the corresponding computational result of the error coding approach, which is the check sequence or frame check sequence (FCS). A codeword can be corrupted by inverting one or more bits in it. These bit corruptions can affect the dataword, the FCS, or both. The corrupted codeword can then be checked to see if it is a valid codeword. In other words, after corruption, there is an error check to see whether the error-detection computation, when performed upon the corrupted dataword, generates a value that matches the corrupted FCS. If the corrupted FCS value from the codeword correctly matches the error-coding calculation performed on the corrupted dataword, the error is undetectable using that error-coding scheme. For an error to be undetectable in codes considered in this work, the dataword must be corrupted, but the FCS may or may not be corrupted to create a codeword with an undetectable error.

Relevant parameters in determining error-detection capabilities of a particular coding scheme include the size of the dataword, the size of the FCS, the number of erroneous bits, and, in some cases, the dataword value before corruption.

Another fault model discussed is that of a burst error, in which two or more bits within a given portion of the codeword are corrupted. For example, a 32-bit burst error has two or more bits in error within a span of 32 bits of the corrupted dataword. Error codes have a burst error detection capability, but, in that usage, detectable burst errors are smaller than or equal to the maximum detectable burst error size. For example, if an error code has 32-bit burst error detection capability, it can also detect all 31-bit, 30-bit, and smaller burst errors.

There are other fault models possible, such as bit slip in communication networks, in which a bit is effectively replicated or deleted from the middle of a bit stream. Those fault models are beyond the scope of this report, although an example of a fault model related to the bit-slip fault is discussed in section 5.2.

2.2 ERROR DETECTION PERFORMANCE CRITERIA.

A study of various error-detection performance criteria mentioned in the literature was performed (see section 7 for a more thorough discussion of sources), and numerous possible criteria were identified. The potential criteria include:

2.2.1 Coding Design Parameters.

- The FCS size 8, 16, 24, 32 bits or other
- Maximum dataword size to be protected and dataword size probability distribution
- Error type: (bisymmetric burst, erasure burst, bit error ratio (BER) bisymmetric, BER random erasure, random data, leading zeros, bit insertion/deletion, bit slip)
- Coding type: (Parity, exclusive OR [XOR], Add, one's complement Add, Fletcher, Adler, Aeronautical Telecommunication Network [ATN] checksum, CRC, cryptographic hash function, multiple CRCs, checksum+CRC)
- Bit encoding: return-to-zero (RZ), non-return-to-zero (NRZ), 8B/10B, bit stuff, start/stop bits
- Interacting design considerations: {bit slip, stuff bits, length field protection, value imbalance}
- Seed values and similar (e.g., to avoid all-zero codewords or to mitigate masquerade attacks)

2.2.2 Potential Evaluation Criteria.

- Hamming distance (HD)
- Hamming weights (HWs)

- Probability of undetected error (Pud) for specified BER and dataword length
- Burst error detection (longest burst for which all errors are caught)
- Burst error performance (HD for a given burst length)
- Shortest dataword size for undetected *k*-bit error (especially $k = 2$ and $k = 5$)
- Multi-HD performance (high HD at short length and lower HD at long lengths)
- Effects of data values on coding effectiveness (applies only to checksums)
- Vulnerability to secondary effects (e.g., unprotected header with length; Controller Area Network (CAN) bit stuff issue)
- Synergy or conflict between error coding and bit encoding
- Computational cost: (speed, memory, recomputing after small dataword change)
- Polynomial divisibility by $(x+1)$
- Use of a primitive polynomial
- Use of published or standardized CRC
- Use of a polynomial with a specific factorization
- Error detection assessment given purely random corruption
- Detection of permutations in data order as the mode of corruption

2.2.3 Application Attributes.

- Configuration audit and configuration management (verifying version of installed software)
- Configuration data integrity check (e.g., message schedule, operating parameters in electrically erasable programmable read-only memory)
- Image integrity check for software update
- Boot-up integrity check of program image (e.g., flash memory)
- Random access memory (RAM) value integrity check

- Network packet integrity check
- Field-programmable gate array configuration integrity check
- Masquerade fault detection
- Implicit group membership check

2.2.4 Other Factors.

- Does system take credit for the following: (OK to have a small number of corrupted values [i.e., system inherently does not malfunction in response to a small number of corrupted values], acknowledgements, repetition, sequence numbers, other end-to-end mitigation strategies)
- Cryptographic authentication, secrecy, and/or integrity mechanisms
- Use of compression
- Temporal failures
- Propagation of message values through multiple paths

2.3 THE ADOPTED FAULT MODEL.

The fault model adopted for this work consisted of random independent bit errors, with a number of other factors addressed in the same context.

One way to evaluate error-detection performance is to consider unconstrained random data corruption. However, for this fault model, the choice of error-detection code is largely irrelevant. If essentially random data are being checked by an error-detection code, then one can expect a chance matching between dataword and *k*-bit FCS with probability 1/(2^*k*), regardless of the error coding approach (in essence, providing the performance of a random hash function as discussed in section 3.1, but due to data randomness rather than good hashing). As the number of faults in a codeword becomes small, there are very distinct differences between coding approaches. That is, the more bits that are corrupted, the less it matters which error-coding approach is used. However, if there are cases in which only a few bits might be in error, then the type of error code is relevant. Thus, selection should be focused on performance for only a few bits when that is a plausible data-corruption scenario. Performance with large numbers of bit errors can be expected to approach random hash performance with increasing numbers of bit errors.

For small numbers of bit errors, the HD of a code becomes relevant. Every code has specific patterns of relatively small numbers of bit errors that it cannot detect. Therefore, if the fault model is a specific pattern of bit errors, code performance depends on that specific pattern, which is not generalizable. However, in most avionics cases, bit errors are not patterned, but are created by noise, random corruptions, or other mechanisms that can be considered random independent events. This random independent error event assumption is the fault model chosen. If a particular system has patterns of bit errors, such as those caused by hardware fault in an intermediate stage in a network, then the general results in this report may not apply.

The question arises as to how often random independent bit errors occur. Because this is a study of error detection rather than error correction, it must be assumed that any erroneous data must be discarded and that the emphasis is on maximizing the probability that data errors are detected. However, for any such system to work, the error rate must be relatively low or essentially all data will be discarded and the system will not perform any useful function. Therefore, it is reasonable for many applications to assume that the anticipated arrival of bit errors is, on average, significantly less than once per codeword (i.e., most codewords are error-free). This concept can be captured in a general way by considering bit errors to arrive via a Poisson process (exponential interarrival time) with a mean of a BER value that is significantly less than the inverse of the dataword size. For example, if codewords are 1000 bits long, the BER should be significantly less than 1/1000 to ensure that most codewords are error free. Given this approach, most codewords are error free, some contain 1-bit error, a few contain 2-bit errors, even fewer contain 3-bit errors, etc. This can be called a BER fault model (also known as a random independent binary symmetric bit inversion model).

As an approximation, the probability of r+1 bit errors occurring in a codeword is BER times the probability of r bit errors occurring. As a result, every bit of HD increase in an error-detection scheme results in an approximate factor of 1/BER reduction in the Pud. A more rigorous treatment of this idea follows.

The probability of an undetected error given a BER fault model depends on the following factors: the BER value, the size of the codeword, the HD of the error code being used, the Pud at the HD, and, in rare cases, the Pud at HD + 1. The Pud at the HD simply means the Pud given that HD bit errors have occurred.

Mathematically, this can be expressed (for a codeword having c bits) as:

$$\text{Pud} = \text{Pud}_{(\text{HD})} + \text{Pud}_{(\text{HD+1})} \tag{1}$$

$$\begin{aligned}\text{Pud} = \text{combin}(c, \text{HD}) * \text{UndetectedFraction}_{\text{HD}} * \text{BER}^{\text{HD}} * (1 - \text{BER})^{(c - \text{HD})} + \\ \text{combin}(c, \text{HD} + 1) * \text{UndetectedFraction}_{\text{HD} + 1} * (\text{BER}^{(\text{HD} + 1)} * (1 - \text{BER})^{(c - \text{HD} + 1)}\end{aligned} \tag{2}$$

In interpreting the first half of the equation, “combin()” is the number of combinations of the c bits of the codeword size taken one HD at a time, which is the total number of possible errors that can occur involving exactly the value of HD erroneous bits. The undetected fraction is the number of undetected errors out of all possible errors that involve HD bits, which varies depending upon the error detection code and how many error bits have been inserted (for a CRC, this is the HW divided by total number of possible errors; for other error-detection codes, the value used is typically the fraction of undetected errors found by a Monte Carlo simulation). The rest of the terms for HD involve the probability that exactly the value of HD bits are in error and exactly (*c*-HD) bits are not in error. The second half of the right side of the equation repeats in a similar manner for HD + 1 error bits.

The Pud result is not affected by fewer than the value of HD errors because 100% of such errors are detected, based on the definition of HD. Terms involving HD + 2 and so on can be neglected because of the assumption that BER is small compared to the codeword size. The HD + 1 term only comes into play when a very small fraction of bit errors involving HD bits are undetected and the UndetectedFraction$_{(HD + 1)}$ is on the order of 1/BER or larger. For many cases, the HD + 1 term does not matter; the cases that do matter tend to be only at codeword lengths near the change of achieved HD, where there are only a few undetected errors having the new, lower HD number of bits in error. Based on this formulation of Pud, the dominant factors in error-detection effectiveness are the undetected error fraction and the HD, with each bit of HD giving an improvement in error detection effectiveness of an additional factor of approximately the BER. For example, with an undetected error fraction of 1.5e-6 (typical of a good 16-bit CRC), a BER of 1e-8, and HD = 4, each bit of HD accounts for almost two orders of magnitude more error detection effectiveness than the undetected error fraction. Thus, while the undetected error fraction is important to consider, for HD greater than 2 (depending on the quality of bit mixing in the error-detection code), HD is usually the dominant factor in achieved error-detection effectiveness. A more detailed example of how to apply these ideas is given in section 6.

Another common selection criterion is computational performance, with checksums often presumed to be calculated much faster than CRCs. However, research into CRC performance has not only produced fast CRC calculation techniques, but also CRC polynomials optimized for computation speed with only minimal reduction of error-detection effectiveness. A research report [3] provides further details. This report considers only error-detection effectiveness. If a checksum or quick-to-compute function provides sufficient error-detection effectiveness, then that may help with computation speed. However, computation speed should not be an excuse to accept inadequate error-detection capabilities.

The following sections describe the evaluation of checksums and CRC in terms of HD, undetected error fraction, and Pud given an assumed BER.

2.4 CHECKSUM EVALUATION.

Checksum error-detection effectiveness evaluation was performed via Monte Carlo simulation using the Mersenne Twister random number generator [4] as the source of pseudo-random numbers. The data array used by that generator was seeded with values from the Unix "random()" linear congruential generator, which was, in turn, seeded with date and time information.

Each trial was performed using the following steps:

1. Generate a random dataword value with an exact predetermined number of randomly positioned bits set to "1" with the rest set to "0."

2. Compute the checksum of this randomly generated dataword and append that FCS value to form a codeword.

3. Generate a random error vector with an exact predetermined number of randomly placed "1" error bits across the entire codeword, with "0" bits elsewhere in the error vector.

4. Compute the corrupted codeword by XORing the error vector into the codeword, resulting in bit flips where the error vector has "1" bits.

5. Determine if the resultant corrupted codeword was valid by computing the checksum of the corrupted dataword and comparing that result to see if it matches the corrupted FCS from the codeword.

6. Record an undetected error if the corrupted codeword was valid (because this means that the corrupted dataword corresponds to the corrupted FCS, resulting in a valid corrupted codeword and, therefore, an undetected error).

Note that the trial procedure differs from simply generating unconstrained random dataword values in that an exact number of desired bits are set in the dataword for every trial (e.g., exactly 50% "1" bits), rather than that number of bits being set on average with some distribution of actual numbers of one bits. Therefore, if 50% of bits are supposed to be 1 bits for a 128-bit dataword, then exactly 64 randomly positioned bits are 1 for each and every trial. Additionally, error bits affect the entire codeword, including dataword, the FCS bits, or both, depending on the error bit placement in a particular trial.

Multiple trials were performed per data point, with a data point being a particular number of error bits and a particular dataword length of interest. Except where noted elsewhere, each data point had enough trials run to experience at least 500 undetected errors for every data point that was used to graph results (and, for all but the longest datawords examined, at least 1000 undetected errors were seen). After confirming that a 50% mix of ones and zeros results in worst case undetected error fractions for checksums [5], datawords with 50% randomly positioned one bit sets were used for all reported Monte Carlo experiments.

Determining the transition point between HD values was difficult because of the need to use random sampling rather than exhaustive enumeration (exhaustive enumeration is impracticable because there are too many possible combinations of dataword values and bit error patterns to consider). Thus, the transition point from one HD to another was estimated using a general approach of finding dataword lengths straddling the point at which undetected errors suddenly disappear for a given sample size. This approach is useful because, based on experimental data, the proportion of error patterns that is undetected varies only slightly as dataword length increases. The transition between HD = 4 and HD = 3 serves as an example. The HD = 3 results (undetected 3-bit errors) were readily found for ATN-32 down to a dataword length of 4064 bits. At that length, a large number of trials were used to evaluate the ATN-32 checksum at a dataword length of 4064 bits with exactly 2032 dataword bits set to "1" and exactly 3 error bits set in each trial. Enough trials were run to find approximately 1000 undetected 3-bit errors, confirming HD = 3 at that dataword length. More precisely, 9.95e11 trials resulted in the discovery of 1064 undetected errors. This same experiment was repeated for the next shorter considered dataword length of 4032 bits (which is one 32-bit word shorter than 4064 bits). At that slightly shorter length, approximately the same number of trials were run (1.01e12 trials), without discovering any undetected 3-bit errors, suggesting HD = 4. While it is always possible that there is still some undetected 3-bit error pattern at this dataword length of 4032 bits, one expects that the density of such undetected error patterns must be dramatically lower than the density of undetected errors at 4064 bits. If the normal behavior of nearby HD = 3 dataword lengths having a similar proportion of undetected 3-bit error patterns held, one would expect to find hundreds of undetected errors at a dataword length of 4032 bits, rather than zero such undetected errors. Put another way, consider two neighboring dataword lengths. The slightly longer one has about 1000 undetected 3-bit errors, while the slightly shorter one has zero undetected 3-bit errors for a comparable number of random trials. Based on that, it is very likely that the boundary between HD = 3 and HD = 4 lies between those two dataword lengths (4-bit undetected errors were identified with a separate set of experiments at a dataword length of 4032 bits, confirming that HD = 4 rather than some higher HD).

While it is tempting to apply statistical analysis to give a confidence interval to these simulation results, the reality is either that there are or are not one or more undetectable 3-bit errors at that length; stochastic simulation either will or will not find one of them. The statistical significance would apply to whether the simulation result of no undetected errors might be correct, but that is not the same as yielding the Pud that would be needed as part of a safety case. It would address the question as to whether the HD is likely to be correct, but it is unclear how to use that number in computing Pud. Given that the ratio of undetected errors is approximately 1 in every 1e9 corrupted codewords for 4064 bits (and less for 4032 bits), it would be easy to miss undetected errors even if a statistical analysis was performed.

One could assume that undetected errors are present even if not found via nonexhaustive simulation, and could apply statistical analysis to estimate the Pud at the HD of bit errors that have not been found, but might possibly be present. That approach is used in an informal manner in section 3.8 when analyzing the ATN-32 HD thresholds. However, coming up with a usable probability requires careful statistical evaluation that is beyond the scope of this report. One factor to consider is that patterns that are undetected by checksums and CRCs are a combination of both pseudo-random error patterns and repeating error patterns. For example, in a checksum with all zero or all one dataword values, an error pattern that consists of only dataword bit flips will be found to occur multiple times as an identical bit pattern starting at every possible dataword position. This is because dataword-only undetected errors are self-contained error patterns in which the contributions of earlier error bits to the checksum calculation are cancelled-out by later error bits, leading to a zero contribution to the final FCS value. This cancellation effect happens regardless of the positioning of the error pattern within the dataword, as long as it does not spill into the FCS region. For example, if a particular error pattern involving 100 dataword bits (and no FCS bits) is undetectable, then a 1000-bit dataword will have 900 such identical patterns of undetected errors, starting at 900 different dataword positions (i.e., the 100-bit undetected error pattern starting at the first bit, that same pattern starting at the second bit, then starting at the third bit, etc. until starting at the 900th bit, after which the error pattern would spill into the FCS and, in most cases, would then be detectable). Thus, any statistical analysis would have to consider the effect of nonrandom distributions of undetected error patterns, which may or may not matter (a similar effect occurs with CRCs, but in that case is independent of dataword values). Moreover, the result would have a statistical confidence level as to the Pud value at not just the presumed HD, but for all numbers of bit errors for which an exhaustive search has not been performed (e.g., if an exhaustive search for 3-bit errors has not been performed, then HD = 3 must be assumed to be true with some Puds even though no cases of undetected 3-bit, 4-bit, or 5-bit errors have been found). Any such confidence levels would have to be taken into account in any system-level analysis. Given that exhaustive searches for checksums require consideration of all possible dataword values as well as all possible error patterns, simulation is generally unsuitable to make definitive statements about HD beyond any HD that can be argued analytically. Thus, both Pud values and HD values based solely on simulations are subject to some level of inaccuracy (CRC performance values for both HD and Pud given in this report are analytically exact, because they consider all possible error patterns and are independent of dataword values).

Returning to the discussion of ATN-32 performance, it seems unlikely (in a difficult-to-quantify sense) that the HD = 4 length is shorter than 4032 bits, especially if one looks at the characteristic downward curves in the plots just above that length (each point forming the curve right at the HD break point is based on 271–622 undetected errors, thus providing a reasonable degree of certainty as to the plotted values). In the absence of an analytic proof of HD characteristics of ATN-32, relying upon HD = 4 performance at dataword lengths up to and perhaps including 4032 bits could be reasonable, depending on the context.

Similar processes were used for other checksums and other lengths, with all other HD breakpoints having at least 1000 undetected errors' worth of trials. There is no analytically exact evaluation method available because checksum performance not only depends upon the error detection computation and the bits in error, but also the data values in the message.

Validity of the checksum computations was checked via comparison to a previous publication [5], which in turn was checked via comparison to known published results. There are no known published results for ATN-32 error-detection capabilities. Therefore, those results should be checked by an independent researcher.

It is possible that, in some applications, purely random dataword values are not representative. That is why the worst case of a 50% mix of ones and zeros is used. There may be specific data value patterns that are even worse because of the particular arrangement of ones and zeros. However, whether or not the difference is significant is not known. Such patterns could potentially change the undetected error fraction, but would not be expected to make the HD worse.

2.5 THE CRC EVALUATION.

The CRC error-detection performance was evaluated using analytically exact exhaustive analyses of all possible *k*-bit codeword error patterns (i.e., for evaluating performance with *k* error bits). The CRC HWs are independent of the dataword values, meaning that dataword values do not matter in this evaluation. The approaches described in references 2 and 6 were used for this analysis. The outputs of the CRC evaluation tool have previously been extensively checked for validity against a wide variety of other CRC error-detection publications.

2.6 OTHER EVALUATIONS.

Evaluations of other specific error-detection technique combinations were done via Monte Carlo analysis, including evaluation of multi-CRC approaches, such as used in Aeronautical Radio, Incorporated (ARINC)-825. There are no known publications for comparison of the other results. However, in some cases, it is straightforward to argue why a particular HD result is valid.

3. CHECKSUM AND CRC ERROR DETECTION.

The error detection performance of checksums depends on how well the checksum computation mixes the bits of data to form an FCS value and whether that mixture is done in such a way that a particular HD is guaranteed for every possible error pattern below that HD value. Sections 3.1 through 3.9 describe error-detection performance for a variety of checksum and CRC approaches.

3.1 RANDOM HASH HD = 1 ERROR DETECTION.

Summary:
Quasi-random mixing function with HD = 1 (e.g., a cryptographically secure hash function)
HD = 1
Undetected Error Fraction at HD: $1/(2^k)$ for k-bit FCS
Burst Error Coverage: None
Is it Data Dependent?: Theoretically; however, mixing quality is independent of data
Does it Detect Change in Data Order?: Yes, subject to HD and undetected error fraction

This is an abstract approach to describing checksum and CRC error-detection performance. The assumption is that some checksum-like function has been defined, which provides excellent mixing of bits in computing a checksum, but it misses detection of some 1-bit errors. In this case HD = 1, but because of the assumption of perfectly random mixing, a change in any 1 bit of the dataword changes each bit of the checksum with an equal and completely random probability. In such a system, it is possible that a 1-bit error in a dataword will have no effect on the FCS value (if HDs higher than 1 were guaranteed, that property would form a predictable pattern in the output of the mixing function, leading, at least theoretically, to a security vulnerability). For this function, the undetected error fraction of 1-bit errors is $1/(2^k)$ for a k-bit FCS. In general, there would be no guarantee of burst error detection coverage because a 1-bit error in the wrong situation may be undetected. It must be emphasized that this is a theoretical limit rather than a finding that applies to any particular hash function. Table 1 shows the theoretical performance of a random hash function.

Table 1. Random Hash Error Detection

Checksum Size Dataword Length = 1024	Analytic Undetected Fraction at HD = 1
8-bit hash	0.00391
16-bit hash	1.53e-5
32-bit hash	2.33e-10

In practice, cryptographically secure hash functions are likely to provide this type of error detection because they are not designed to provide a certain HD, but instead cryptographically secure mixing of bit values.

3.2 PARITY.

Summary:
Single parity bit added to dataword
HD = 2
Undetected Error Fraction at HD: 100% within a dataword
Burst Error Coverage: None
Is it Data Dependent?: No
Does it Detect Change in Data Order?: No

Single-bit parity is computed by XORing all the bits of a dataword to form one single parity bit. That bit is a zero if the number of “1” bits in the dataword is even, and is a one if the number of “1” bits in the dataword is odd (some systems invert the sense of the parity bit so that an all zero dataword has a parity value of 1, avoiding the possibility of an all zero codeword).

Per-word parity gives HD = 2 on a per-dataword basis with no burst error detection. Any 2-bit errors in a dataword plus parity data set are undetected and performance is insensitive to data values. Because it is a per-word error code, each word has error detection independent from other words. It also costs one extra bit per dataword, making it more expensive in terms of bits than other error codes for all but the shortest datawords.

All even-number bit errors are undetected by parity, so the undetected error fraction at HD = 2 is 100% within a dataword. When considering a multiword message, the analysis depends on data length. However, the general outcome is that, for an error to be undetected, all words must either be error-free or have an even number of errors (see section 4.1 for a case study). Thus, the undetected error fraction is less than 100% for 4-bit errors and other higher-order even numbers of bit errors because of the decreasing probability that all errors will be allocated in pairs to words of a multiword message.

3.3 LONGITUDINAL REDUNDANCY CHECK.

Summary:
Block-wise XOR of all data blocks
HD = 2
Undetected Error Fraction at HD: 3.125% for 32-bit chunk size at all dataword lengths
Burst Error Coverage: Chunk size
Is it Data Dependent?: No
Does it Detect Change in Data Order?: No

A longitudinal redundancy check (LRC), also known as an XOR checksum, involves XORing all the chunks of a dataword together to create a check sequence. It is equivalent to computing the parity of each bit position within a chunk independently (i.e., check sequence position *i* is the parity of bit position *i* across all chunks in the dataword). There is no obvious seminal publication for LRC, but it is referred to frequently in the checksum literature.

The LRC error codes give HD = 2, with burst error detection up to the chunk size. Any 2-bit errors in the same bit position of a block (including the FCS) are undetected, and error detection performance is insensitive to data values. If any bit position has an odd number of bit errors, the LRC will detect that there is an error, improving average case performance compared to a single parity bit. An LRC can be thought of as computing a separate parity bit for each bit position of the block, and saving those parity bits as an FCS. An error is detected unless every one of those parity bits fails to detect all errors in that bit position.

The probability of an undetected 2-bit error for LRC depends on the computation chunk size. Conceptually, it is the probability that the 2-bit errors happen to align in the same bit position of the chunk (e.g., for an 8-bit chunk with one error in bit 3 of the chunk, the second error has to be in bit 3 of some other chunk for the FCS to be undetected).

As per Maxino and Koopman [5], the probability of an undetected 2-bit error is:

$$\frac{n-k}{k(n-1)} \tag{3}$$

In this equation, *k* is the number of FCS bits, and *n* is the number of codeword bits. Undetected error probabilities from Monte Carlo Simulation with a 1024-bit dataword are given in table 2, with the analytic values from the Maxino equation.

Table 2. The LRC Error Detection

Checksum Size Dataword Length = 1024	Simulated Undetected Fraction at HD = 2	Analytic Undetected Fraction at HD = 2	Difference From Analytic
8-bit LRC	0.12417	0.12415	0.01% worse
16-bit LRC	0.06160	0.06160	0.00%
32-bit LRC	0.03032	0.03033	0.04% worse

Figure 1 shows simulation results for a 32-bit LRC and other checksums on a log–log plot, with dataword lengths in bytes. Plots for 8-bit and 16-bit checksums are similar. While the undetected error fraction is constant, the probability of multibit errors increases with dataword length, resulting in a corresponding increase in Pud.

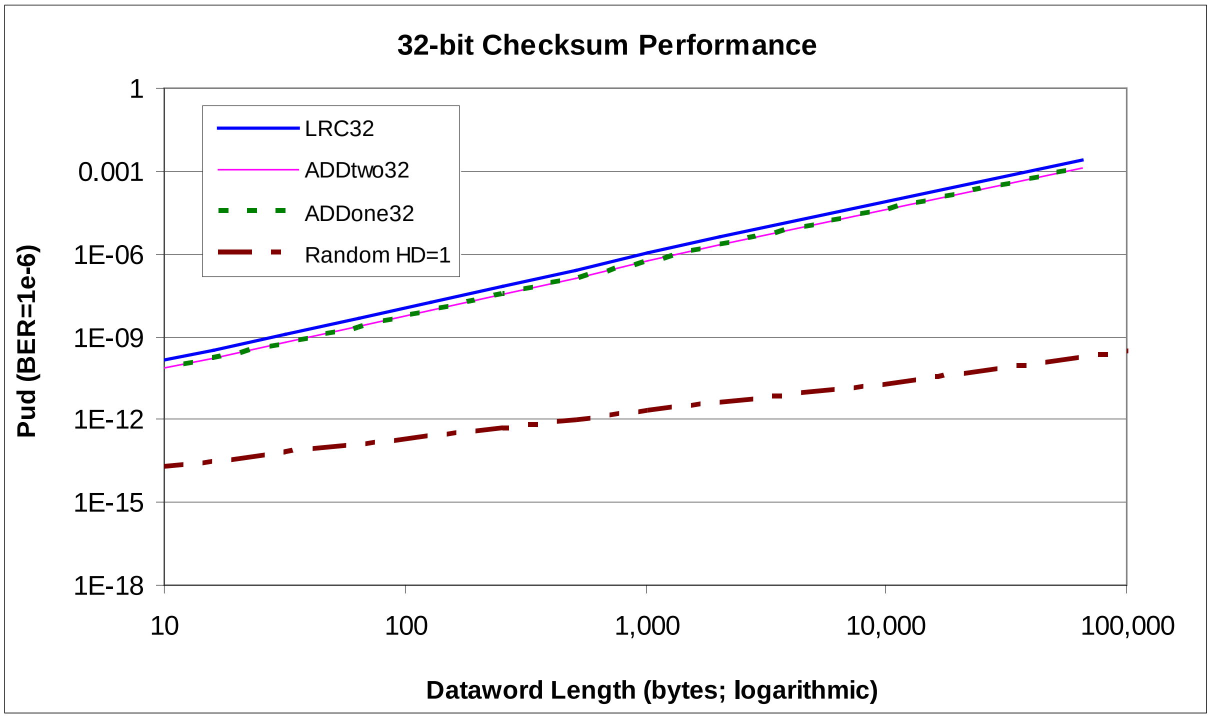


Figure 1. The Pud for LRC and Addition Checksums (downward means better Pud)

Of note in figure 1, an HD = 1 random hash value does better than an HD = 2 checksum. This is because the undetected error fraction of the checksums shown is quite high, due to relatively poor bit mixing by the checksum functions. In other words, a good hash function with HD = 1 can be expected to do better than one of these checksums with HD = 2 (however, other checksums and CRCs can do much better than the checksums shown in this figure).

When plotting error-detection performance, as in figure 1, it is important to be mindful of the assumed BER. The values plotted will depend heavily upon that value. Additionally, it is in most cases meaningless to extend the graph past the point at which every message is likely to have an error; the system using error detection codes will throw away essentially all of its data as erroneous. In the case of figure 1, the longest length plotted is 100,000 bytes = 800,000 bits, which is close to BER = one error per 1,000,000 bits.

3.4 TWO’S COMPLEMENT CHECKSUM.

Summary:
Normal integer addition of all data blocks
HD = 2
Undetected Error Fraction at HD: 1.565% for 32-bit FCS
Burst Error Coverage: Chunk size
Is it Data Dependent?: Yes
Does it Detect Change in Data Order?: No

Checksums are computed by using integer addition of all chunks of a dataword, with the resultant sum forming the check sequence. As with parity-based computations, it is difficult to define a seminal publication for two’s complement addition checksums.

A two’s complement checksum gives HD = 2 with burst error detection up to the chunk size. It operates by using normal (two’s complement) addition instructions. The number of 2-bit errors that are undetected is reduced compared to an LRC error code because a pair of inverted bits in the same bit position can sometimes be detected by the fact that the carry generated by adding those bits is different. However, reliance on carry operations to improve checksum effectiveness means that the performance is data dependent, with performance best at either all zero or all one dataword values and worst with a random 50% mix of one and zero dataword values.

As is typical for two’s complement computer arithmetic, carry-outs from the highest bit of the addition are ignored, making the top bit of each block more vulnerable to undetected errors (all bits in the block are protected by the carries from an addition operation, but the topmost bit in the block is protected only by an LRC operation due to throwing away the carry-out from the topmost bit addition).

The analytic undetected error fraction is complex, but a simplified approximation that is most accurate at long dataword lengths and pessimistic at short data lengths is given by Maxino and Koopman [5]:

$$\frac{k+1}{2k^2} \tag{4}$$

Table 3 shows two’s complement error-detection performance. Figure 1 shows that a two’s complement checksum is significantly better than an LRC due to the internal carry bits reducing the probability of undetected 2-bit errors. However, the significant amount (approximately a factor of two) is still relatively small on a log–log scale.

Table 3. Two's Complement Checksum Error Detection

Checksum Size Dataword length = 1024	Simulated Undetected Fraction at HD = 2	Published Undetected Fraction Approximation at HD = 2 [5] (valid for all lengths)	Difference
8-bit chk2	0.06990	0.07031	0.59%
16-bit chk2	0.03275	0.03320	1.36%
32-bit chk2	0.01565	0.01611	2.86%

3.5 ONE'S COMPLEMENT CHECKSUM.

Summary:
One's complement integer addition of all data blocks
HD = 2
Undetected Error Fraction at HD: 1.518% for 32-bit FCS
Burst Error Coverage: Chunk size
Is it Data Dependent?: Yes
Does it Detect Change in Data Order?: No

In terms of the mechanics of computation, a one's complement addition is the same as a two's complement addition, except that in the case of the one's complement, the carry-out of each addition is added back into the lowest bit position of that addition result. In other words, if a two's complement addition results in a carry-out of zero, then that is used as the one's complement addition result. However, if a two's complement addition results in a carry-out of one, then the addition result is incremented by one to give a one's complement addition result. This is an implementation method that correctly deals with the fact that an all zero bit pattern and an all one bit pattern both represent the number zero in one's complement arithmetic.

From an error-detection point of view, the benefit of this computational approach is that carry-out bits are captured during the computation instead of being thrown away, slightly increasing bit mixing and error detection effectiveness. Usas provided analysis demonstrating the superiority of one's complement addition to two's complement addition [7].

The analytic undetected error fraction is complex, but as simplified approximation for long data lengths is given as in Maxino and Koopman [5]:

$$\frac{1}{2k} \tag{5}$$

Table 4 shows one's complement error detection performance. Figure 1 shows that, whereas a one's complement checksum provides slightly better error detection than a two's complement checksum, the difference is difficult to see on a log–log plot. If possible, a one's complement checksum should be used rather than a two's complement checksum. However, it is better to use a more advanced checksum, such as a Fletcher checksum.

Table 4. One's Complement Checksum Error Detection

Checksum Size Dataword length = 1024	Simulated Undetected Fraction at HD = 2	Published Undetected Fraction Approximation at HD = 2 [5]	Difference
8-bit chk1	0.06214	0.06250	0.58%
16-bit chk1	0.03083	0.03125	1.34%
32-bit chk1	0.01518	0.01563	2.86%

3.6 FLETCHER CHECKSUM.

Summary:
One's complement integer addition of two running sums
HD = 3 for short lengths; HD = 2 for longer lengths
Burst Error Coverage: At least half of FCS size (i.e., size of one running sum)
Is it Data Dependent?: Yes
Does it Detect Change in Data Order: Yes, subject to HD and undetected error fraction

The Fletcher checksum [8] is computed with a pair of running arithmetic sums. The first running sum is simply the one's complement sum of all chunks in the dataword. The first running sum is added into the second running sum after every data chunk is processed:

$$SumA = SumA + \text{next data chunk} \quad (6)$$

$$SumB = SumB + SumA \quad (7)$$

Both *SumA* and *SumB* are initialized to some constant value. The additions are one's complement addition and the process is iterated over data chunks for the entire dataword with the data chunk size being half of the FCS size. At the end of the checksum computation, the FCS is the concatenation of *SumA* and *SumB*. Thus, a 32-bit Fletcher checksum is the concatenation of a 16-bit *SumA* value and a 16-bit *SumB* value. This iterative approach leaves the second sum with one times the last data chunk, two times the next-to-last data chunk, three times the next-most-recent data chunk, etc. This makes it possible to detect changes in data order for HD = 3 error-detection lengths. Fletcher checksums use one's complement addition as previously described for a one's complement checksum.

Nakassis [9] provides implementation hints for the Fletcher checksum and provides error detection effectiveness metrics based on probability of undetected random corruption, 16-bit burst errors, single-bit errors, and minimum distance between undetected double-bit errors. The incorrect use of two's complement addition instead of one's complement addition for computing a Fletcher checksum is shown to reduce the minimum distance between undetected double-bit errors from 2040 bits distance for a 16-bit Fletcher checksum (two 8-bit chunks) to only 16 bits distance. Nakassis also suggests a metric of ease of recomputing a checksum after a small change to the dataword and suggests that the Fletcher checksum is better according to that metric.

An 8-bit Fletcher Checksum has HD = 3 up to 60 dataword bits and HD = 2 above that, which this report has confirmed [5]. A 16-bit Fletcher checksum is HD = 3 up to 2039 bits, which this report has confirmed. Maxino and Koopman [5] similarly projected HD = 3 up to 1,048,559 bits, which this report has found to be incorrect after extensive simulation. No 2-bit undetected error was found at that length with 1e8 trials of Monte Carlo simulation, which run quite slowly because of the large data structures involved. However, based on the patterns of Fletcher checksum computation, there is reason to believe such a pattern exists and the predicted HD = 2 dataword length. However, it could take well in excess of 1e9 trials to detect one, which would exceed the time available given fixed computational resources.

One way to look at the Fletcher checksum performance pattern is to consider that the running sums are half the FCS size. In those terms, an 8-bit Fletcher checksum has HD = 2 performance starting at just over 15 dataword blocks of 4 bits. A 16-bit Fletcher checksum has HD = 2 performance starting at 255 dataword blocks of 8 bits and a 32-bit Fletcher checksum has HD = 2 performance starting at 65,535 dataword blocks of 16 bits.

Table 5 shows that Fletcher checksums achieve a better HD than addition checksums up to a certain length determined by the size of the Fletcher checksum.

Table 5. Fletcher Checksum Error Detection

Checksum Size		Simulated Undetected Fraction of 2-Bit Errors	Simulated Undetected Fraction of 3-Bit Errors
8-bit Fletcher:	(56 bits)	0	0.00388
	(64 bits)	0.00153	0.00381
	(1024 bits)	0.00780	0.00317
16-bit Fletcher:	(1024 bits)	0	5.28e-5
	(2032 bits)	0	4.76e-5
	(2048 bits)	3.76e-6	4.81e-5
32-bit Fletcher:	(1024 bits)	0	2.77e-5
	(65536 bits)	0	3.35e-7
	(1048544 bits)	Should be 0	>0 (insufficient samples to provide exact number)
	(1048576 bits)	>0(none found)	>0 (not measured)

Figure 2 shows that Fletcher checksum performance is significantly better than other checksums discussed and has good enough mixing properties that it is better than a random hash for all short-to-medium dataword lengths (up to approximately 8 kilobytes [Kbytes]), but worse for long dataword lengths. This is because the addition operation gives the Fletcher checksum only mediocre mixing. It has HD = 3 instead of HD = 2 performance at those lengths and the extra bit of HD makes up for the poor mixing for this BER value. Note that in this report figures show dataword lengths in bytes and tables show dataword lengths in bits.

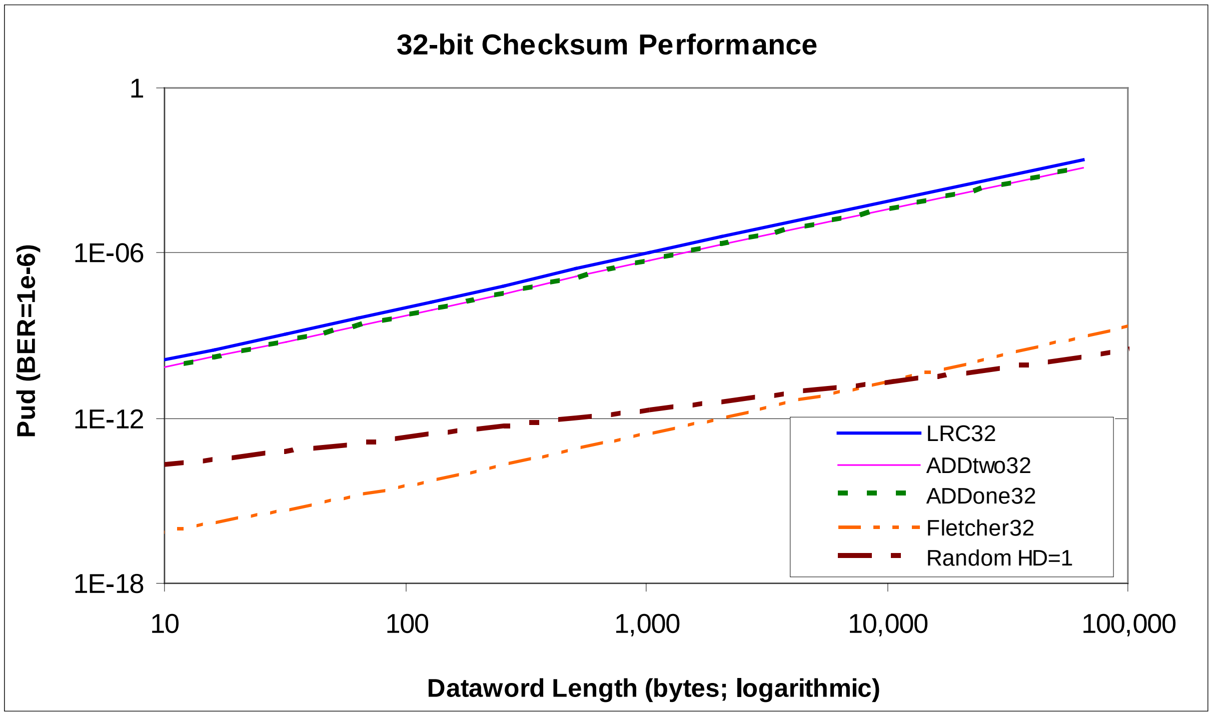


Figure 2. The 32-Bit Checksum Performance

Figure 3 shows a comparison of 8-bit, 16-bit, and 32-bit Fletcher checksum performance. The dramatic decreases in Pud for the 8-bit and 16-bit curves correspond to the places where they degrade from HD = 3 to HD = 2 performance. This change is significantly larger than the distance between curves where they have the same HD, emphasizing the importance of HD in error detection given a BER fault model.

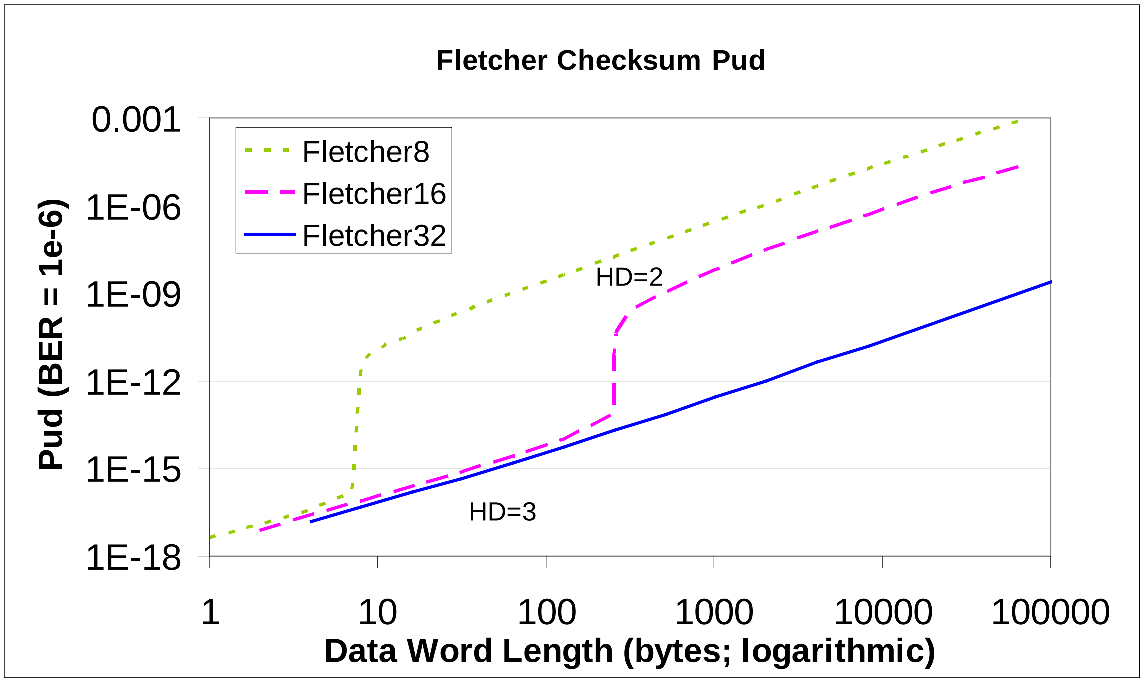


Figure 3. Fletcher Checksum Performance

3.7 ADLER CHECKSUM.

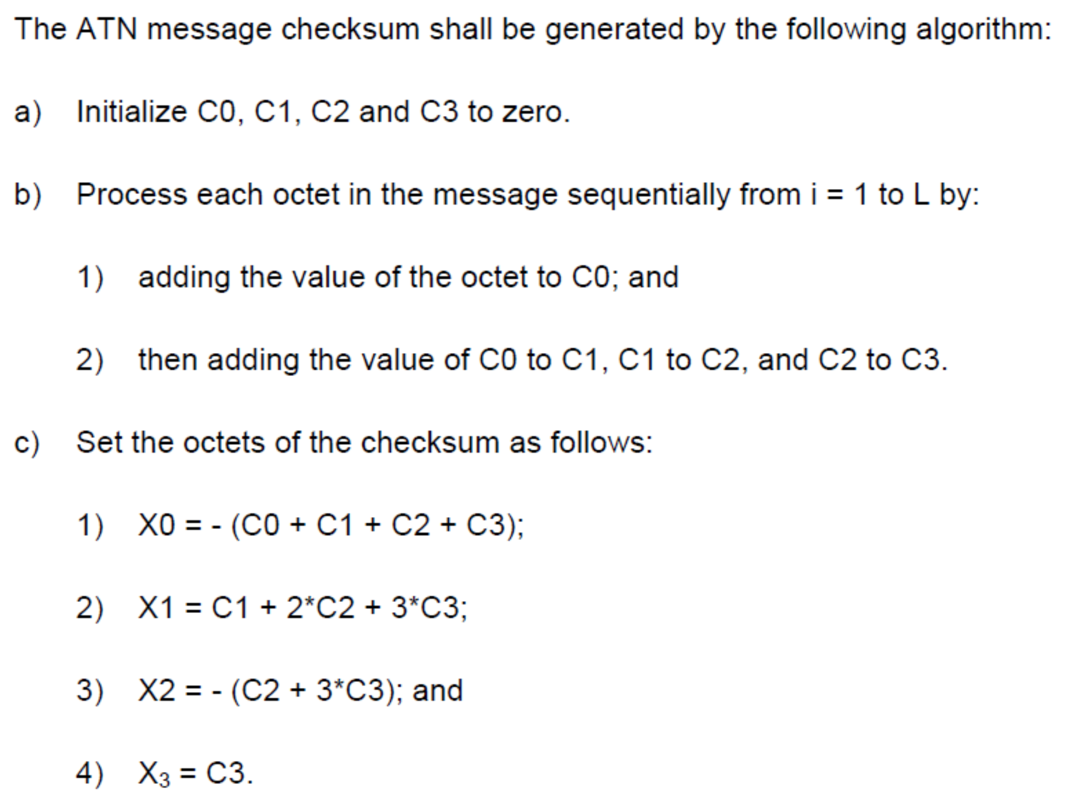
The Adler checksum [10] is the same as a Fletcher checksum, except Adler checksums use modulo addition. The modulus chosen is the largest prime number smaller than the relevant power of 2 for the chunk size being added. In practice, this means an 8-bit chunk size is added modulo 251 (i.e., using a modulus of 251), and a 16-bit chunk size is added modulo 65,521.

The use of a prime modulus provides better mixing of bits, but comes at the cost of having fewer valid check sequence values. This is because some FCS values are never generated because of the modular division. For example, with an 8-bit chunk size, the values 251, 252, 253, 254, and 255 are invalid under the modulus of 251 and, therefore, cannot appear in either byte of the 16-bit check sequence, giving fewer possible check sequence values and slightly increasing the probability of a multi-bit error creating a valid codeword randomly. The tradeoff of better mixing with a prime modulus versus fewer possible check sequence values was studied [5]. In general, an Adler checksum usually is worse than a Fletcher checksum and only improves error detection by a slight amount in the regions where it is better. Because of the more expensive computation required, this checksum was not studied further and, in general, is not recommended as a worthwhile tradeoff. Given the computational complexity required, a CRC should be used instead of an Adler checksum.

3.8 THE ATN-32.

Summary:
ATN-32 modified Fletcher checksum
HD = 3, 4, or 6 for shorter lengths
HD = 2 for long lengths
Burst Error Coverage: At least one quarter chunk size
Is it Data Dependent?: Yes
Does it Detect Change in Data Order?: Yes, subject to HD and undetected error fraction

The ATN-32 checksum can be thought of as a modified Fletcher checksum that uses four running one’s complement sums of 8 bits each to create a 32-bit checksum value [11]. The final value is a computational mixture of the four running sums rather than a simple concatenation, as shown in figure 4 (a reproduction of figure 6.25 of reference 11).



The ATN message checksum shall be generated by the following algorithm:

a) Initialize C0, C1, C2 and C3 to zero.

b) Process each octet in the message sequentially from i = 1 to L by:

   1) adding the value of the octet to C0; and

   2) then adding the value of C0 to C1, C1 to C2, and C2 to C3.

c) Set the octets of the checksum as follows:

   1) X0 = - (C0 + C1 + C2 + C3);

   2) X1 = C1 + 2*C2 + 3*C3;

   3) X2 = - (C2 + 3*C3); and

   4) X_3 = C3.

Figure 4. The ATN-32 Checksum Algorithm [11]

Error detection effectiveness of the ATN-32 checksum is shown below. It has HD = 6 up to 96-bit datawords, HD = 4 up to 4032-bit datawords, and HD = 3 up to 6080-bit datawords. At and beyond 6112 bit datawords, it has HD = 2. This gives an additional bit of HD up to 4032 bits compared to a conventional Fletcher checksum, but, as a tradeoff, degrades to HD = 2 at only 6112 bits.

It should be noted that ATN-32 applications assume use to well beyond 6112 bits of dataword length because it is designed to be used with Internet Protocol (IP) messages [11]. This means that the maximum message size in the absence of system-specific constraints can range up to approximately 1500 bytes, or 12,000 bits, with almost that many bits needing to be protected by the ATN-32 checksum. However, HD = 2 is all that is achieved above 6112 bits (764 bytes). Thus, applications should not rely upon an HD of greater than 2 unless they limit message lengths accordingly.

Table 6 shows ATN-32 checksum performance. The "//" entries indicate that simulation data is expected to be non-zero, but was not measured with enough accuracy to be reported.

Table 6. The ATN-32 Checksum Error Detection

ATN-32 Dataword Size	2-Bit Errors	3-Bit Errors	4-Bit Errors	5-Bit Errors	6-Bit Errors
96 bits	0	0	0	0	3.19e-9
128 bits	0	0	5.74e-8	//	//
4032 bits	0	0	1.34e-7	//	//
4064 bits	0	1.25e-9	//	//	//
6080 bits	0	4.13e-8	//	//	//
6112 bits	6.27e-7	//	//	//	//

Figure 5 shows the error-detection performance of ATN-32. Note that ATN-32 is significantly better than a 32-bit Fletcher checksum at short lengths, but degrades to HD = 2 and is, therefore, significantly worse at and above 764 bytes (6112 bits) of dataword length. Note that because an analytic analysis of HD is not available, any reliance on HD greater than 2 is suspect, and must at least be accompanied by a statistical analysis of the probability of HD being less than the figure shows, as discussed in section 2.4.

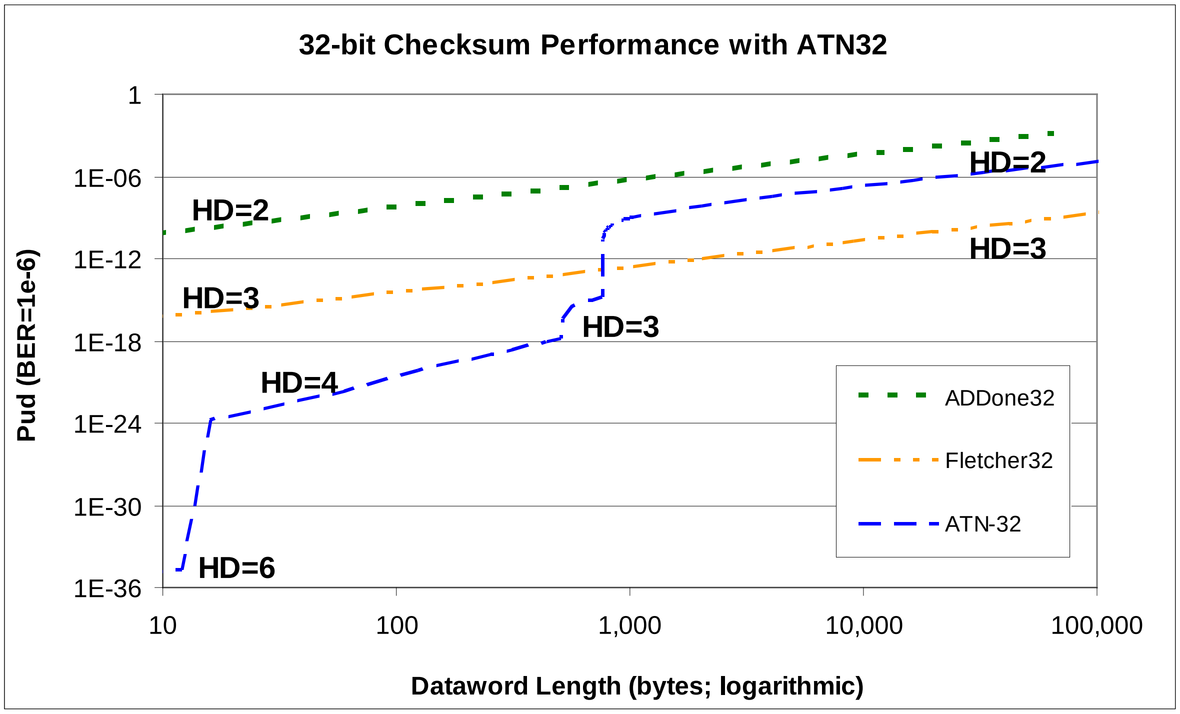


Figure 5. The 32-Bit Checksum Compared to ATN-32 Performance

3.9 THE CRC.

Summary:
The CRC computation
HD: Varies based on feedback polynomial
HD = 2 for long lengths
HD for shorter lengths depends upon specifics of polynomial and dataword length
Burst Error Coverage: Chunk size
Is it Data Dependent?: No
Does it Detect Change in Data Order?: Yes, subject to HD and undetected error fraction

The CRCs are mathematically based on polynomial division over Galois field (2). This means that the dataword is considered to be a polynomial in which each bit has an ascending power of variable "*x*" with a "1" or "0" coefficient, depending upon the binary value of the dataword at that bit position. A "CRC polynomial" is used to divide the dataword and the remainder is used as the FCS value. This division process can be performed via a shift-and-XOR procedure for each bit of the dataword, but much more efficient implementation methods are in use (e.g., Ray and Koopman [3]). While a CRC computation may be slower than a checksum computation, it can produce considerably better error-detection results. The error-detection properties are largely determined by the particular CRC polynomial used.

The CRCs have some error-detection properties that are worth noting. Error detection for the BER fault model is completely independent of dataword values (i.e., unlikely checksums, the value of the dataword has no effect on error-detection capabilities). The error-detection capability for the bit-reverse of a CRC polynomial is identical to the original CRC polynomial, even though the actual value computed is different.

Figure 6 shows Institute of Electrical and Electronics Engineers (IEEE) Standard 802.3 CRC-32 error-detection performance compared to checksums. It achieves better HD than all types of checksums shown, with HD = 4 shown for large lengths on that figure. Other CRCs may perform dramatically better (reaching up to HD = 6 for datawords as long as maximum-length Ethernet messages and even better HD possible for shorter datawords).

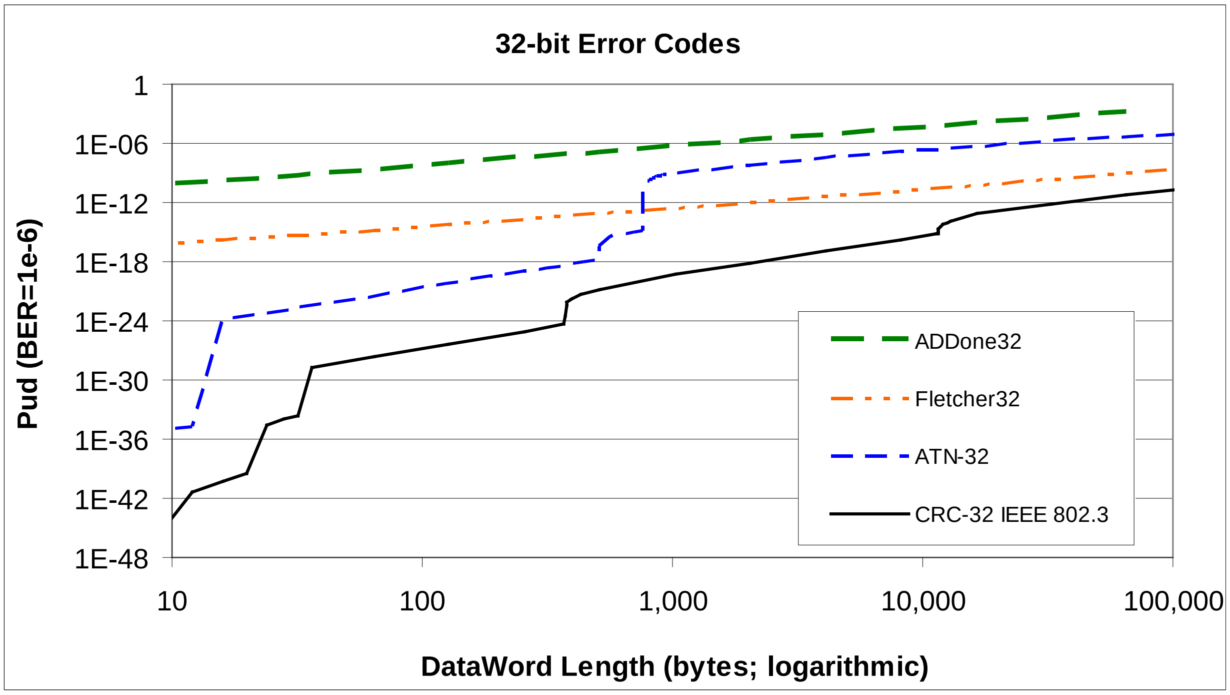


Figure 6. A 32-Bit Error Code Performance Including CRC-32

Table 7 shows generically good CRC polynomials for a variety of HD values. By “good,” it is meant that they have a particular HD out to the longest dataword length of any polynomial of that size (found via exhaustive search) and, additionally, have even higher HD at shorter dataword lengths. The lengths shown are the maximum dataword length for which that polynomial achieves the stated HD. The number shown in a column is the highest number of data bits for that HD. This table shows some useful polynomials that are in implicit +1 notation (there are a number of different notations in use). Thus, for example, 0xA6 has HD = 4 up to 15 bits, HD = 3 up to 247 bits, and HD = 2 at all lengths at and above 248 bits.

Table 7. Error Detection of Good CRC Polynomials (lengths in bits)

Polynomial	HD = 2	HD = 3	HD = 4	HD = 5	HD = 6
0xA6	Good	247	15		
0x97	Longer	--	119		
0x9C	Longer	--	--	9	
0x8D95	Good	65,519	1149	62	19
0xD175	Longer	--	32,751	--	54
0xBAAD	Longer	--	7985	108	20
0xAC9A	Longer	--	--	241	35
0xC86C	Longer	--	--	--	135
0x80000D	Good	16,777,212	5815	509	
0x9945B1	Longer	--	8,388,604	--	822
0x98FF8C	Longer	--	--	4073	228
0xBD80DE	Longer	--	4074	--	2026
0x80000057 [3]	Good	4,294,967,292	--	2770	325
0x80002B8D [3]	Longer	--	2,147,483,646	--	3526
0x8F6E37A0 [6]	Longer	--	2,147,483,646	--	5243
0xBA0DC66B [6]	Longer	--	114,663	--	16,360
0xD419CC15	Longer	--	--	65,505	1060
0x90022004 [6]	Longer	--	65,506	--	32,738
0x82608EDB (CRC-32)	Longer	4,294,967,292	91,607	2974	268

Dataword lengths of less than 8 bits are omitted from table 7, as is the presence of any HD > 6 lengths. The notation "--" means that dataword lengths are not provided for that HD. The notation "good" means it is a good polynomial to use for long datawords. The notation "longer" means HD = 2 performance at dataword lengths longer than the highest number in other columns for that polynomial, and that the performance is not as good as the "good" polynomial at that size for long dataword lengths. All CRC performance numbers are analytically exact in this table and in the analysis, except where use of an approximation in the analysis is explicitly stated.

For situations in which many bits of corruption are expected, the approximate undetected error fraction for large numbers of bit errors is relevant. In general, CRCs provide superior performance to all types of checksums for large numbers of bit errors because they provide superior bit mixing along with good HD. This characteristic was not specifically measured across all CRCs because of computational infeasibility for many error codes studied. However, based on previous work examining many different CRCs, in general, the undetected error fraction for CRCs converges to $1/((2^k)-1)$ for large numbers of bit errors (the “-1” term is because any non-zero value cannot produce an FCS of zero, reducing the number of possible FCS values by 1 for every dataword value except all zeros). The CRCs are almost as good as a perfect hash function, except CRCs have the advantage of HD = 2 or better at all dataword lengths. The exception is that for CRC polynomials divisible by $(x + 1)$, all odd number of bit errors are detected, but at the price of a corresponding doubling of the undetected error fraction for even numbers of bit errors to approximately $1/(2^{(k-1)})$.

3.10 OTHER ERROR-DETECTION CONSIDERATIONS.

Three other error detection considerations appear in sections 3.10.1 through 3.10.3.

3.10.1 Effect of Initial Seed Values.

Checksums and CRCs involve a repeated computation over the length of a dataword that accumulates a result to form an FCS value. The accumulation value must be initialized to some known value for this process to produce a predictable result for any given codeword. When the initial value is unstated, it is typically assumed to be zero.

In some cases, a non-zero value is used instead, such as 0xFFFF for a 16-bit checksum or CRC. This has the advantage of making an all zero codeword impossible. Perhaps more importantly for Fletcher checksum, ATN-32, and CRCs, this has the additional advantage of causing leading zeros of a message to affect the FCS value. To better understand, consider a CRC initialized with an all zero seed value. Any number of all zero dataword chunks can be fed into the CRC and it will stay at a zero value, throwing away information about how many zero chunks have been processed. However, if the CRC computation is initialized to any non-zero value, processing chunks with a value of zero will cycle patterns through the CRC, tracking the number of zero-valued chunks that have been processed as a sort of counter. This can help with situations in which additional leading-zero bytes have been erroneously added to a dataword.

In general, any non-zero seed value can be used. The value selected does not affect error-detection properties beyond those previously discussed. In particular, CRC error detection, in terms of the BER fault model, is independent of the seed value used.

The CRC and checksum values can be alternately or additionally modified to create an FCS value. A common modification is to XOR the result of a checksum computation with a value, such as 0xFFFF, which inverts bits, but does not lose any of the information content of the checksum computation. In some cases, CRC result values must be bit-reversed to preserve burst error properties, depending upon the computation method and message bit ordering in the dataword.

3.10.2 Burst Error Detection.

All checksums and CRCs detect all burst errors up to the data chunk size (note that in the case of Fletcher and ATN-32 checksums, the data chunk size is less than the FCS size). However, this property depends on the correct bit ordering of computation in two ways. First, it is assumed that the bit ordering in the computation is the same as the bit ordering of transmission in a network or physical bit ordering in a memory. Second, it is assumed that the bit ordering of the FCS value is such that the last bit of the protected dataword primarily affected the last bit of the FCS field (i.e., the last bit of the dataword and the lowest order bit of the FCS are as far apart as possible).

A mistake in bit ordering within bytes, data chunks, or the FCS can reduce burst error coverage. Ray and Koopman [3] show that IEEE 1934 Firewire has a 25-bit burst error-detection capability instead of the expected 32-bit burst error-detection capability expected from application of its CRC-32 error-detection code. This is because of a problem with bit ordering of transmitted bytes being the reverse of bit ordering within bytes used for CRC computations. This issue should be taken into account when evaluating the error-detection effectiveness of aviation applications of Firewire.

3.10.3 Detection of Changes in Data Order.

In some applications, it may be desirable to detect changes in the order of data (e.g., when multiple network packets are reassembled into a long dataword). The LRC and additive checksums are insensitive to data order because they simply add the data chunks. Thus, they will never detect an error that is solely in the form of misordered data chunks.

The Fletcher checksum (and the similar ATN-32) will detect most data ordering errors as long as the out-of-order data chunks are recent enough to be within the span of HD = 3 dataword lengths. To understand why this is the case, consider that the *SumB* term of the Fletcher checksum in effect multiplies each data chunk by a constant integer, indicating how far that data chunk is from the FCS. For a 16-bit Fletcher checksum, this multiplication aliases modulo 256, preventing detection of all reordering problems for data exchanges that are a multiple of 256 data chunks apart.

The CRCs in general can detect most data-ordering errors. Any vulnerabilities to data-ordering errors would be specific to the particular data values being sent and the particular polynomial being used, and would be expected to be generally pseudo-random in nature.

In the above discussion, there is no guarantee that all possible data-ordering errors will be detected. Rather, the conclusion to be drawn is that ordinary checksums provide no capability, Fletcher checksums provide some capability, and CRCs arguably provide the best capability to detect data-ordering errors. Quantifying this protection further is beyond the scope of this report.

4. AVIATION-SPECIFIC RESULTS.

The IEEE 802.3 Ethernet CRC-32 polynomial is widely used in all application areas, including aviation. Beyond that observation, some representative aviation protocols were examined. Specifically, ARINC-629 was studied as an example of a mature, purpose-designed aviation network protocol; ARINC-825 was studied as an example of a protocol adapted from the automotive domain to aviation uses; and the ATN-32 checksum was studied as an example of a checksum retrofitted to existing desktop computing technology (Ethernet).

4.1 THE ARINC-629.

The ARINC-629 standard communication protocol was studied as an example of a mature, aviation-specific protocol. This standard permits use of one of either a 16-bit CRC or a 16-bit two's complement addition across a 4096-bit dataword. It also uses per-word parity on every 16-bit machine word transmitted.

Perhaps surprisingly, simulation results showed HD = 4 error-detection coverage when using either the CRC or the checksum (with the CRC giving a better undetected error fraction at HD = 4). The reason is that the parity bit on each 16-bit network transmission word (which is a data chunk for the error-detection code) ensures that all 1-bit errors in a chunk are detected, and that only a 2-bit error in the same transmitted 16-bit data chunk (which is really 16 data bits plus one parity bit) will remain undetected. Because undetected errors for both a CRC and a checksum require the corruption of two data chunks, this requires four different bit errors to result in an undetected error—two bits in a first data chunk to get past the parity check, and two bits in a second data chunk to again get past the parity check.

Thus, the inclusion of per-chunk parity dramatically increases the effectiveness of a checksum, giving HD = 4 at the cost of one extra bit per machine word. This result should apply to any application that combines parity and a checksum (assuming some other effect does not undermine the resulting HD = 4 performance).

4.2 THE ARINC-825.

The ARINC-825 standard communication protocol was studied as an example of a mature communication protocol developed for automotive use and repurposed for aviation use. It is an aviation standardization of the CAN, which is a protocol with several known issues that must be worked around to provide dependable communication (see sections 5.1 and 5.2).

Of particular interest for this study is the CAN's vulnerability to bit-stuff errors. In particular, a pair of bit errors in the bit-stuffed format of a CAN message (i.e., corruption of only two transmission bits as seen on the wire) can cause a cascading of bit errors in the message, leading to 2-bit errors that are undetected by the CRC [12]. This vulnerability had not been acknowledged by the ARINC-825 standard, which stated that the CAN CRC provided a theoretical HD = 6 error detection, but omitted mention of the bit-stuff problem, which gives a practical error-detection capability of HD = 2 (see reference 13, section 4.6.1.3).

The ARINC-825 provides for an additional CRC in the data payload for high-integrity messages, which goes beyond the baseline CAN standard upon which it builds. When the experiments were attempted to quantify the performance of this additional Message Integrity Check (MIC) CRC [13], several typographical errors and ambiguities in that portion of the specification were discovered, making it impossible to successfully implement a conforming CRC implementation.

The authors of this report have been invited to suggest corrections to ARINC-825 to address these issues, which will take place in due course. Meanwhile, Monte Carlo simulation of a draft corrected interpretation of the MIC confirmed that the addition of a second CRC does not improve HD beyond 2, but does reduce the undetected error fraction because any undetected errors have to evade detection by three successive error checks: the bit unstuffing mechanism, the MIC CRC in the data payload, and the normal CAN CRC.

5. FRAMING, ENCODING, AND MULTIPLE CRCs.

This section addresses specifics of data framing, bit encoding, and the use of multiple CRCs to provide error detection. An important theme is that, in some cases, layers that are supposed to be independent from a network perspective (e.g., bit framing and error detection) can interact to degrade error-detection capabilities.

5.1 CORRUPTED LENGTH FIELD.

The way that data is framed in network messages can affect how well CRCs work. One problem is that, if a length field is in a message, a corruption of that length field can cause the receiver to look in the wrong place for the CRC value (see figure 7).

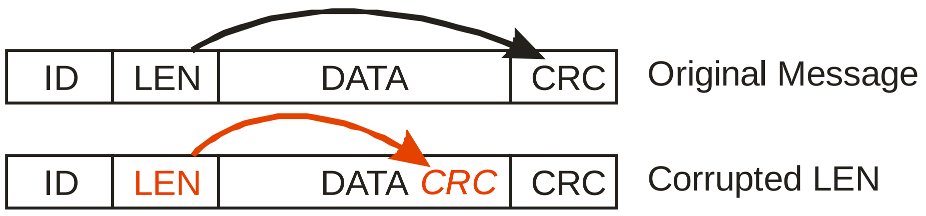


Figure 7. Corruption of Message Length Field

A single bit error can corrupt the length field and result in an apparent CRC value (via accessing part of the dataword that is not really the FCS). This forms a valid codeword at that reduced length (an HD = 1 vulnerability that bypasses CRC protection). This vulnerability is discussed in Driscoll, et al. [14]. Common countermeasures for this vulnerability include providing parity on transmitted data chunks (as implemented by ARINC-629) and assurances that corrupted length fields will be caught via checks on received message length (as stated in the standards for ARINC-629, ARINC-825, and others—although thorough testing and analysis must be performed to ensure that end-of-length checks are sufficiently robust to avoid degrading CRC performance on any particular protocol or implementation).

The automotive FlexRay standard [15] goes further by providing a header CRC independent of the payload CRC. Because the FlexRay header CRC is always in the same place for its fixed header size, it can ensure the integrity of the length field. Once the length field has been protected to a sufficient level, a separate payload CRC protects the rest of the network message.

5.2 BIT STUFF VULNERABILITIES.

As previously described for ARINC-825, bit stuffing creates a vulnerability by which corruption to stuff bits can result in an effective “shift” of bit values seen by the CRC, circumventing the promised HD [12]. Assuming that bit counts are checked, this creates a vulnerability of HD = 2 (one stuff bit must be converted into an apparent data bit, and a second data bit converted to an apparent stuff bit to preserve the number of bits). Effective countermeasures might include computing CRCs on stuffed values, although such an approach would have to be studied in detail to work out issues such as how to bit stuff the CRC value itself. In the absence of a validated solution, it seems prudent to avoid bit stuffing if better error detection than HD = 2 is required.

5.3 BALANCE BIT ENCODING.

Some protocols use bit encoding to ensure a balance of one and zero physical bits. Gigabit Ethernet (1000BASE-X) and 10G Ethernet (10GBASE-X) use 8B/10B encoding. In 8B/10B encoding, each group of 8 data bits is encoded into a set of 10 physically transmitted bits that have approximately the same numbers of ones and zeros to provide a waveform with zero DC bias (in the case of 8B/10B, there is some carry-over of bias from one symbol to the next in the form of a running disparity that is accounted for in the encoding scheme). However, this means that a single physical bit error in the 10-bit physical format has the potential to corrupt multiple dataword bits when the corrupted physical message is mapped from 10B back to 8B format. Other high-speed networks use similar approaches.

A simulation of injecting faults into Ethernet 8B/10B encoded data with the IEEE 802.3 CRC did not find any undetected faults at an HD worse than the usual CRC-32 HD. This is not a conclusive finding because a very large, but limited, number of Monte Carlo simulation runs were performed. This simulation used strict 8B/10B decoding, in which all malformed physical encodings were rejected. However, some hardware in the field uses relaxed decoding [16], which can map invalid 10-bit values to 8-bit data values while ignoring the fact that the 10-bit value is invalid. Both experimental evidence and analytic evidence was found that such problems can be undetected by the Ethernet CRC-32. A definitive answer would require further investigation of the multi-burst properties of CRC-32. In the interim, it would seem wise to require that Ethernet 8B/10B decoders reject any malformed 10B encoding (i.e., that they be strict decoders).

5.4 DATA SCRAMBLERS.

Some data networks, such as faster Ethernet standards, use data scramblers. For example, data scramblers are used in an attempt to break up long runs of zeros that may be produced by software (e.g., unused values or large counter fields with small actual values). The effectiveness of such approaches depends on the details of the system (in some systems, there is a bit pattern that will produce an all zero or all one output when scrambled). However, more important for error detection, scramblers can degrade burst error protection.

For example, 33-bit scramblers are used in some forms of Ethernet. The CRC-32 polynomial used for error detection has a burst error-detection length of 32 bits. However, if errors with a burst length of 32 bits or shorter are passed through a scrambler, those errors can affect 33 bits of the result. While detailed simulations have not been performed, one should assume that there is at least one 32-bit (or shorter) error pattern that can be scrambled into a 33-bit value that is undetected by a 32-bit CRC. Any system with a scrambler that spans more bits than the burst length of the corresponding CRC should be presumed to have its burst length error-detection properties defeated by the scrambler, unless exhaustive analysis has proven otherwise.

5.5 ENCRYPTION.

A detailed study of encryption is beyond the scope of this report, but it should be noted that data encryption can be looked at as a type of sophisticated scrambler that potentially reaches over the entire length of the dataword. A single error bit in an encrypted data stream can affect an unlimited number of bits in the encrypted stream (depending on the type of encryption used). Thus, if an error detection code is encrypted along with its dataword, it should be presumed to have an effective HD = 1 error-detection capability (a single bit error in the encrypted dataword can result in an error pattern that exceeds the checksum or CRC's effective HD).

5.6 MEMORY DATA PLACEMENT EFFECTS.

For memory error-detection applications, there may be concern that a single fault can affect bits that are on adjacent rows or columns, which are quite far apart in terms of their apparent location in the dataword. For example, a radiation particle strike might affect four memory cells if it hits midway between two rows and midway between two columns (cells to upper-left, upper-right, lower-left, and lower-right of the strike point near the four-corners cell position). Not much can be said about this in general because number of bits in each row, interleaving, and spare rows/columns of the memory greatly affect the precise bit patterns that may be created by such a fault.

There is one CRC property that may be helpful in such situations. If a CRC has an HD up to a particular dataword length, then that HD will hold for any bit errors within that length, even if the protected dataword itself is significantly longer. For example, consider an 8 Kbyte dataword protected by an IEEE CRC-32. That CRC has HD = 6 up to 268 bits. If the memory has a row size of 256 bits and is not interleaved, then an error that hits the corner of four physically adjacent bits will be detected in all cases. That is because four bits must, by construction, be within 258 bits of each other in the apparent dataword accessed from that memory. A CRC-32 detects any possible 5-bit error pattern within 268 bits, so it will detect this error if there are no additional errors elsewhere in the dataword (one additional error bit still keeps the HD below 6, but if the burst size is greater than 268 bits, the error pattern might not be detected, even if four of five of the errors are that close together). This property might be significant motivation to adopt a 32-bit CRC with a better HD = 6 or HD = 5 length than CRC-32, such as 0x90022004, which provides HD = 6 up to nearly 32 kilobits.

5.7 MULTIPLE CHECKSUMS AND CRCs.

It is common to use multiple checksums, CRCs, parity fields, or combinations of these to improve error-detection capabilities. Several examples are described in this report (e.g., sections 4.1, 4.2, 7.7.1, and 7.7.2). Based on the results of this study and a literature review, there is no evidence that multiple error codes on the same dataword increase HD. However, adding more bits of error-detection code does tend to increase the coverage at that HD by creating more bits that must be randomly matched to form a valid codeword. Quantifying such a gain requires careful analysis of common-mode undetected error patterns for the multiple error coding schemes used, with a caution that pure mathematical analysis is prone to subtle errors and should always be validated via Monte Carlo simulation.

A special case exists for which adding multiple checksums or CRCs can be shown to improve HD. This takes place when one error code is used to protect a small dataword and another error code is used to protect a larger dataword composed of many small, protected datawords. The improvement of effective HD of ARINC-629 because of the use of parity is an example. Again, careful analysis is required to understand any gains, and that analysis should always be validated via Monte Carlo simulation. A special case is the use of small dataword CRCs combined with an overarching CRC for memory.

6. MAPPING CRITICALITY TO INTEGRITY COVERAGE.

6.1 IMPORTANCE OF COVERAGE DETERMINATION.

One of the most important considerations for selecting an error-detection scheme is the degree of coverage required. Because these schemes consume resources, such as processor time, memory space, and/or communication bandwidth, avionics designers do not want to use a scheme with expensive overkill, which would needlessly increases size, weight, and/or power of an avionics system.

In addition to traditional aspects of cost, there can be safety tradeoffs with the addition of an error-detection scheme. As with almost all fault tolerance mechanisms, there is a tradeoff between availability and integrity. That is, techniques that increase integrity tend to reduce availability and vice versa. Employing error detection by adding a check sequence to a dataword increases integrity, but decreases availability. The decrease in availability happens through false-positive detections. These failures preclude the use of some data that otherwise would not have been rejected had it not been for the addition of error-detection coding.

False-positive detections can occur in two primary ways. The first occurs because the check sequence adds more bits to the dataword and each bit that is added is another bit that could fail. For checksums and CRCs, a failure in the check sequence bits is indistinguishable from failures in the dataword. Thus, a failure in the check sequence will cause the data to be flagged as failed, which takes place because of the addition of the check sequence. Thus, the addition of the check sequence bits can cause the lack of availability of the dataword to which it is attached. The other way that false positives can occur is a failure in the mechanism (hardware or software) that implements the scheme. This can be due to a design flaw and/or failure of the mechanism after it has been fielded (e.g., a “stuck at bad” error detector). Again, an error-detection mechanism that is itself faulty and indicates that good data is bad causes a loss of availability for that data, which would not happen if that mechanism did not exist.

In addition to the false positives problem, another way that the addition of an error-detection scheme can reduce safety is through opportunity costs. An opportunity cost problem is one in which the error detection scheme consumes resources that otherwise could be used to improve safety in a different way. For example, the processor time needed to generate and check the check sequence for a dataword could reduce safety margins by increasing data-processing latency. Although such an occurrence is very unlikely, avionics designers should make sure that adding an error-detection scheme does not reduce safety through an opportunity cost.

Because of such considerations, avionics designers should select an error-detection scheme that is not excessive, but still provides sufficient coverage. Thus, it is important to determine accurately what the sufficient level of coverage is. The following subsections describe a process for determining a sufficient level of coverage.

6.2 SCOPE.

The places where checksums and CRCs may be used in avionics systems can be divided into three areas. These areas and their characteristics include:

1. Error coding for data-transfer systems—some error-detection coding guidance exists (Federal Aviation Administration [FAA] Advisory Circular [AC] 20-156). Such systems can be characterized as:

    a. Dynamic; data is moving
    b. Including onboard data networks and off-board communication
    c. Providing very general guidance

2. Error coding for memory—while no specific guidance exists, standards such as ARINC 665, RTCA, Inc. (RTCA) DO-200A, and RTCA DO-201A describe some error-detection methods. Such systems can be characterized as:

    a. Static; data is at rest; can be volatile (RAM) or non-volatile (flash, PROM)
    b. Having typically hierarchical memory
    c. Possibly flash-copied to RAM
    d. Possibly RAM-copied to and from processor cache (several levels)
    e. Not having specific guidance for error-detection coding requirements

3. Error coding for transferable media—while no specific guidance exists, standards such as ARINC 665, RTCA DO-200A, and RTCA DO-201A describe some error-detection methods. Such systems can be characterized as:

    a. More like memory, even though data is moving

    b. For example, magnetic disks, optical discs, and flash devices (via Universal Serial Bus, Secure Digital cards, etc.)

    c. Not having specific guidance for error-detection coding requirements

Within each of these areas, the process described for mapping criticality to coverage pertains only to checksums and CRCs that protect against errors created in the media. For data-transfer systems, the media comprises all passive components through which messages travel (e.g., wires, fiber, and connectors). Media coverage does not necessarily include complex electronics, such as switches and routers, which are too complex to characterize in terms of faulty behavior coverage of checksums or CRCs.

For memory devices, the medium is the memory device itself. The CRCs and checksums do not necessarily provide coverage for a processor that reads this memory nor any complex electronics that may be in the path between the processor and the memory.

For transferable media, checksums and CRCs cover only errors that occur within the media itself and do not necessarily cover any complex electronics that may read or write that media.

In the following subsections, the phrase "data-transfer system" is used to mean one or more data networks and/or any other mechanisms that transfer data into avionics via messaging. The phrase "storage system" is any type of memory or combination of memories (including flash, RAM, cache, disks, etc.). The phrase "data-transfer or storage system" is the union of the two previous phrase definitions.

6.3 DATA-TRANSFER SYSTEM GUIDANCE BACKGROUND.

One area in which the FAA has given some guidance for error detection and where it is likely that checksums or CRCs would provide that detection is in the area of data buses. The text of FAA AC 20-156 "Aviation Databus Assurance" [17] that pertains to databus error tolerance is the beginning of its paragraph 4:

> 4. DATA INTEGRITY REQUIREMENTS. Data that passes between LRUs, nodes, switches, modules, or other entities must remain accurate and meet the data integrity of the aircraft functions supported by the databus. To ensure data integrity through error detection and corrections resulting from databus architectures, evaluate databus systems for the following:
>
> a. The maximum error rate per byte expected for data transmission.
>
> b. Where applicable, databus-provided means to detect and recover from failures and errors to meet the required safety, availability and integrity requirements. Examples are cyclic redundancy checks, built-in detection, hardware mechanisms, and architecture provisions.

Though this wording appears to be specific to onboard databus systems, this guidance should be applicable to any data-transfer systems, including off-board connections, such as radio channels and "sneaker net" transferable media. The AC 20-156 provides general guidance. The process for determining an adequate level of integrity coverage described in the following subsections is much more detailed.

6.4 MAPPING PROCESS OVERVIEW.

This section describes one approach to determining an adequate level of integrity coverage using CRCs and checksums. Other approaches may also be feasible as long as they propose a concrete and defensible explanation as to how all the elements of this approach are addressed.

Figure 8 shows an overall process flow for determining an adequate level of integrity coverage. Each step in the process is tagged with a letter in a circle to reference each tagged step in the following text. In the following subsections, each step of this flow is described in more detail. The description of each step includes how the step would be applicable to an example avionics data-transfer system. The same example system is used throughout the description of each one of the steps.

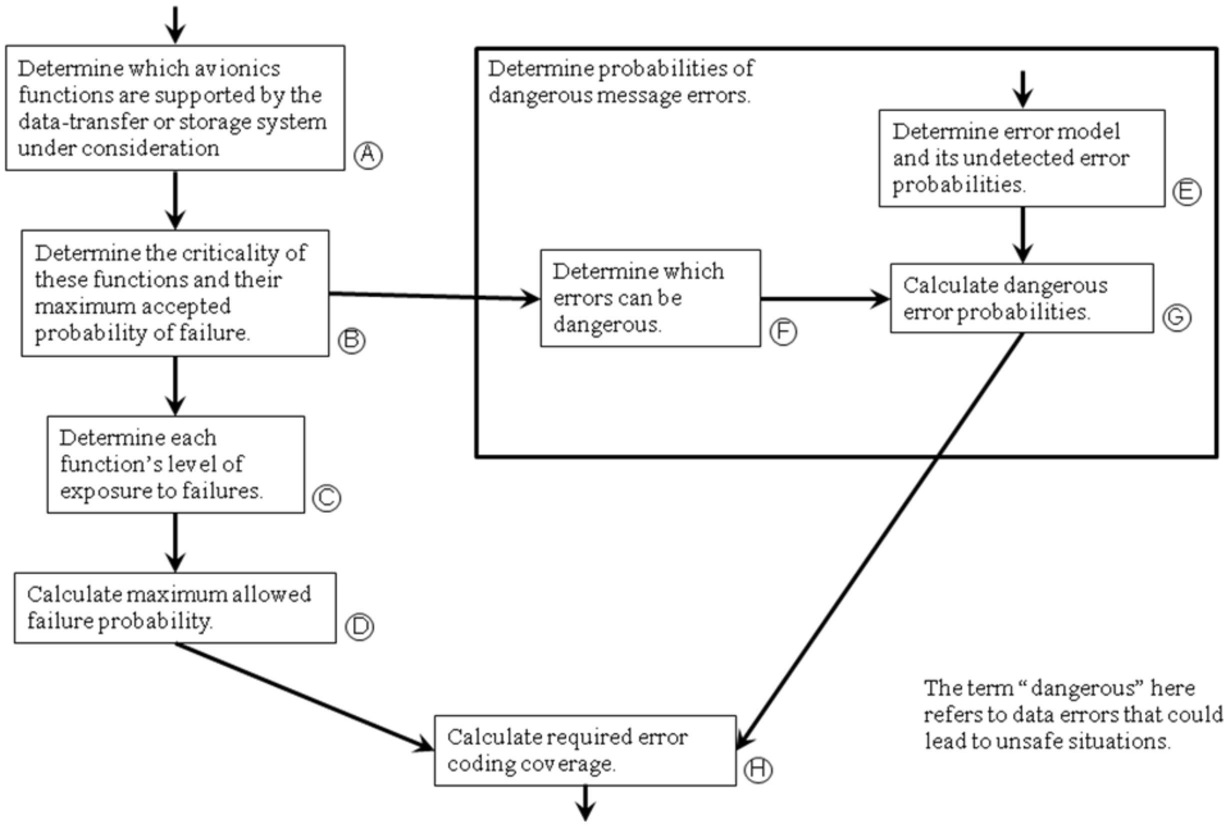

Determine which avionics functions are supported by the data-transfer or storage system under consideration
A
Determine the criticality of these functions and their maximum accepted probability of failure.
B
Determine each function's level of exposure to failures.
C
Calculate maximum allowed failure probability.
D
Determine probabilities of dangerous message errors.
Determine error model and its undetected error probabilities.
E
Determine which errors can be dangerous.
F
Calculate dangerous error probabilities.
G
Calculate required error coding coverage.
H
The term "dangerous" here refers to data errors that could lead to unsafe situations.

Figure 8. Criticality Mapping Flow Diagram

6.5 EXAMPLE AVIONICS DATA-TRANSFER SYSTEM.

A fictitious data-transfer system was created as an example for this report, so as not to represent any existing or planned system. The example system needs to be somewhat unrealistic to ensure that it is not like any planned system (this is the consequence for ensuring that no one can read into this report any accidental similarity between the example system and an actual system).

The example system includes an autopilot control panel that uses a redundant databus network to talk to a set of autopilot computers. This example artificially[*] increases the criticality of this function by ignoring the fact that pilots are required to know and maintain assigned altitudes,

[*] Some may argue this increase is valid, because some pilots rely on the autopilot when they should not.

independent of the autopilot. Further details about this example system are revealed in the following subsections.

6.6 CRITICALITY MAPPING FLOW, STEPS A AND B.

The first two steps for determining the coverage requirements are obvious and are part of any avionics design process that follows accepted practices. Step A determines which avionics functions are supported by any data-transfer or storage system. Step B determines the criticality of these functions and their maximum accepted probability of failure for each identified data-transfer or storage system.

6.6.1 Criticality Mapping Flow, Steps A and B Example.

During the execution of Functional Hazard Assessment and System Safety Assessment early in the design cycle of this example system, it was found that messages from the control panel to the computers send altitude-hold changes as increments or decrements to the existing altitude-hold value. It was also found that an undetected error in these increments or decrements could cause an aircraft to try to hold an incorrect altitude. This could lead to a midair collision, a condition that cannot be allowed with a probability of more than 1e-9 per flight hour (the catastrophic failure probability limit per FAA AC 25.1309).

6.7 CRITICALITY MAPPING FLOW, STEP C.

Once identification has been made for all of the critical functions that are supported by data-transfer or storage systems, the next step is to determine each function's level of exposure to failures. That is, an analysis must be done to determine the worst-case hazard that could result from any erroneous behavior of the data-transfer or storage system. This analysis can be done with the help of well-known design methodologies and tools, such as fault trees and Markoff models. In addition to identifying the worst-case scenario, this analysis can provide probability information to be used in step D.

6.7.1 Criticality Mapping Flow, Step C Example.

Because this system uses increments or decrements to the existing altitude-hold value rather than re-sending an absolute value for the altitude hold, any undetected transient error in a databus message transfer can cause a permanent error in the autopilot's altitude hold value. The latter cannot be allowed to happen with a probability of more than 1e-9 per flight-hour. Thus, the system must ensure that undetected transient errors in the databus system do not exceed this probability. The system has redundancy mechanisms to deal with missing or rejected messages, but no integrity mechanisms above the network. Therefore, the network exposes the system to integrity failures.

6.8 CRITICALITY MAPPING FLOW, STEP D.

The maximum allowed failure probability was calculated using the information from steps A through C.

6.8.1 Criticality Mapping Flow, Step D Example.

Because this network transfers data that must not fail with a probability of more than 1e-9 per flight hour, and there are no additional integrity mechanisms in the example system, messages must not have undetectable errors with a greater probability (i.e., the maximum acceptable Pud is 1e-9 per flight-hour).

6.9 CRITICALITY MAPPING FLOW, STEP E.

Steps A through D have determined the maximum allowable undetected error probability. Steps E through G will be used to determine the probability that errors will occur. The difference between these two will be used to determine the amount of error-detection coverage needed.

Step E is used to determine the fault model and its undetected error probabilities (that is, the probabilities that errors will be produced and escape detection in the absence of any checksum or CRC mechanism). In addition, this step must determine the characteristics of these errors. For example, questions similar to the following should be asked during analysis for this step:

1. Is the probability of a bit having an error independent of any other bits having an error?

2. If the probability of one bit having an error is correlated to the probability of another bit having an error, what is this correlation?

    a. Are the bits physically adjacent to each other in a memory device?

    b. Are the bits physically (for networks with parallel paths) or temporally adjacent to each other in a communication transfer?

    c. Can the error probability of one bit be influenced by the history of previous bits due to transmission line reflections or DC imbalance droop?

    d. Can compression, encryption, scrambling, symbol encoding, etc. cause error multiplication (i.e., a single bit error in the medium causing multiple bit errors to be perceived by the mechanism doing error detection)?

3. Can the probability of errors change over time (e.g., because of aging or environmental changes)?

    a. If so, what is the worst case?
    b. Are there detection and mitigation strategies for these changes?

It is important to note that some media have error sources that can be characterized and some that cannot. An example of the former is the media of data-transfer systems that typically consist of wires or optical fibers and connectors. The error sources for these components are noise, droop, and reflections. An example of media aspects that cannot be characterized is the precise path taken through all the gates and registers of a complex electronic device. Because the probabilities of various types of errors for these complex electronic devices cannot be determined, one cannot establish what the coverage needs to be for checksums or CRCs.

Designers need to be cautioned not to succumb to the common misconception that the probability for the outcome of an event is the reciprocal of the number of outcomes. For example, one cannot say the probability of three bits being 000 is 1/8, unless one can first show that the probability of each bit being zero is 50% and then additionally show that the probability of each individual bit being zero is totally uncorrelated with the probability of other bits being zero. In typical avionics random messages, the probability of getting a string of zeros is higher than the assumption based on a 50% probability for each bit being 0. This is common in networks because of the convention of setting unused payload fields to zero and leading zeros when standard-size fields, such as bytes, are used to represent integer numeric values with ranges smaller than the full range of that standard-sized field.

It is important for avionics system designers to avoid incorrect assumptions about the randomness of data value distributions and error distributions when it comes to error-detecting codes. This misunderstanding is typically revealed as assuming that all bits fail with an independent 50% probability or that data is a 50% mix of ones and zeros, with no rationale for why this should be the case. In general, neither the independence assumption nor the 50% error probability per bit assumption can be shown to be true for complex electronics devices. However, such assumptions may be justifiable for simpler situations, such as data bits transmitted on a wire if the worst case data value distribution can be determined and used in analysis.

6.9.1 Finding a BER.

For data-transfer systems, the error model is typically characterized by the system's BER. Avionics system designers need to confirm BER by testing. The next few sections describe how to find this BER. While these subsections are directed at data-transfer systems, similar considerations apply to storage systems (e.g., determining memory single-event upset rates).

If the data-transfer system is based on a standard, one cannot simply state the BER requirement in the standard as the BER that the system achieves. What is in the standard is a requirement, not a guarantee. It is also important to note that aircraft are not the same as cars or offices. The cables and connectors used in avionics typically are not per the commercial, off-the-shelf (COTS) standard. Also, the aircraft environment is not the same. Non-avionics standards are required only to meet particular test scenarios; they do not necessarily have to be the worst case. To be economically viable, COTS components need to work only for the mainstream customers. These parts do not have to meet BER requirements for all scenarios. In particular, they typically are not designed to meet the standard’s BER in an avionics environment with a confidence level commensurate with that required by avionics (e.g., AC 25.1309). Thus, all data-transfer systems need to be BER tested. At the very least, this must be done during system tests. These tests may also need to be repeated over the system's lifetime to detect system degradation. These test requirements apply to both COTS and custom components.

6.9.2 The BER Tests.

The BER tests must be performed in the avionics' worst-case environment (e.g., per DO-160). A worst-case bit pattern must be transmitted (e.g., the pattern(s) most likely to result in maximum droop/minimum eye opening). Network standards typically define these patterns. If more than one worst-case bit pattern is defined, all such patterns must be tested. Certain pathological bit patterns (e.g., Ethernet IEEE 100BASE-TX "killer packets") can be ignored if the data-transfer system has mechanisms to prevent them from ever being transmitted. The BER tests must present the worst-case eye pattern (the most degraded signal) to each receiver design. Some test equipment can generate bit patterns with worst-case eyes. If this equipment is not available, testers must build a physical analog for the data-transfer system's worst-case cable. Note that the worst-case cable may not be the longest one, particularly where significant reflections are possible.

Some BER test equipment uses its own internal receivers. Unless there is a known relationship between the performance of that test equipment's receivers and the receivers actually used in the data-transfer system, these results are useless for making claims about the BER in the data-transfer system. Similarly, the results from network analyzers and similar test equipment that estimate BER based on signal-to-noise ratio or eye-opening cannot be used unless there is a well-known relationship between these parameters and the actual performance of the receivers in the data-transfer system.

Given that the BER is the number of bad bits seen by the receiver-under-test divided by the total number of bits sent, the accuracy in the BER number depends on the total number of bits sent. A sufficient number of bits must be transmitted such that the confidence in the BER test results is commensurate with that needed by the avionics' certification plan.

6.9.3 The BER Degradation.

During an aircraft's lifetime, degradation of the data-transfer system's components might increase the BER. This higher BER may overwhelm error detection mechanisms. That is, given that a fraction (about 0.5^k for good k-bit random hash error code) of errors are undetected, the higher BER would give a corresponding increase in the number of undetected failures. For example, a typical undetected error rate could be represented by:

$$\text{undetected\_error\_rate} = \text{bit\_rate} * \text{BER} * 0.5^k \tag{8}$$

This formula makes it clear that the undetected error rate is directly proportional to BER.

One way to determine if data-transfer system component degradation has increased the BER to an unacceptable level is to rerun the BER testing periodically. However, this can become very expensive. An alternative is to do some form of prognostics. An obvious, simple prognostic is to watch the trend in detected errors. An elevated number of detected errors implies an elevated BER. A detection scheme can be based on the expected detected error budget per hour, computed at design time. If this budget is significantly exceeded, one can infer that the BER specification has been violated. In response, the system must stop trusting the data source and data-transfer system because of the high BER. Note that detected errors are only symptoms, and not the problem—undetected errors are the problem. If a data-transfer system is producing a lot of erroneous messages, it is likely that a few messages that happen to pass error checks actually have undetected errors.

6.9.4 Criticality Mapping Flow, Step E Example.

The autopilot control panel is connected via databuses directly to the autopilot computers, with no intervening switches or other complex logic devices. Therefore, the results of BER tests can be used to determine error probabilities. If there were any complex devices in the data path, there would be no way to determine individual bit error probabilities or the probabilities for specific bit error patterns. For this example, test results showing a BER of 1e-8 (which meets the ARINC 664 requirement) will be assumed.

6.10 CRITICALITY MAPPING FLOW, STEP F.

Because not all data bits in a dataword can lead to unsafe situations if they are erroneous, step F is used to determine which errors can be dangerous. However, because of the complexities in determining the coverage for some subset of bits within a dataword, it may be simpler to assume that an error in any of the data bits could lead to a dangerous condition. This simplification is discussed in section 6.15.

6.10.1 Criticality Mapping Flow, Step F Example.

The messages between the autopilot control panel and the autopilot computer are 64 bytes long[*], 4 bits of which are the critical altitude change data. If any of these 4 bits are corrupted and not detected, the aircraft will try to hold an incorrect altitude.

6.11 CRITICALITY MAPPING FLOW, STEP G.

Calculating the dangerous error probabilities uses straightforward probability math.

6.11.1 Criticality Mapping Flow, Step G Example.

The probability that a message has an attitude data error is the same probability that any of the 4 altitude bits is erroneous. With a 1e-8 BER, this probability is:

$$1 - (1 - 1e\text{-}8)^4 \approx 3.9999999400000004e\text{-}8 \tag{9}$$

If this system sends this message at 20 Hz, then 72,000 messages are sent per hour (20 messages/second * 60 seconds/minute * 60 minutes/hour). The probability that one or more instances of this message type have an altitude data error in an hour is:

[*] 64 bytes is the minimum Ethernet frame (message) size—including Ethernet addressing, length/type, and CRC fields, all of which are covered by the CRC.

$$1 - (1 - 3.9999999400000004\text{e-}8)^{72000} \approx 2.8758567928056934555\text{e-}3 \qquad (10)$$

Because this is greater than 1e-^{9}, error detection is needed.

Note that the equation form "$1 - (1 - p)^{n}$" gives the probability that one or more of *n* events, each of independent probability *p*, occur. This can be approximated by *n* * *p* if *p* is small enough. However, this approximation can be used only if one is sure that the loss of accuracy is inconsequential. In this case, one cannot be sure of that, a priori, since the example's failure probability limit is very small (1e-^{9} per flight-hour).

6.12 CRITICALITY MAPPING FLOW, STEP H.

The final step uses all the information gathered in the previous steps to calculate the required error coding coverage. The coverage of an error coding design has to be such that:

> The conditional probabilities of particular error bit patterns (which may depend on BER, number of critical bits, and message length)…x
>
> the probability of the error coding design failing to detect these patterns
>
> < the maximum allowable failure probability for a message.

6.12.1 Criticality Mapping Flow, Step H Example.

Adding the symbol *C* to represent simple coverage (the probability that an error in the four altitude bits will be detected) to the dangerous error probabilities equation from step G and making it less than 1e-^{9} gives:

$$1 - (1 - (3.9999999400000004\text{e-}8 * (1 - C)))^{72000} \leq 1\text{e-}9 \qquad (11)$$

Solving for *C*, gives $C \geq 0.99999965277777239583569847611 9$. Thus, the error detection must have greater than 99.99996527777723958357% coverage on each message. This would be sufficient coverage for an error-detecting code only if all the following were known to be true:

1. The hardware/software that does the error detection never fails
   (Cannot be guaranteed)

2. The error code needs to protect only these 4 bits rather than the whole message
   (Rarely done in practice)

3. The error code itself never suffers any errors
   (Cannot be guaranteed)

The first of these will be covered in section 6.14 with the other two combined and covered in section 6.15.

6.13 ERROR DETECTION MECHANISM FAILURE.

For this example scenario, assume that the error-detection mechanisms' failure rate is 1e-7 per flight-hour. The probability for the error-detection mechanisms being functional for t flight-hours is $e^{-(1e\text{-}7\ *\ t)}$. With scrubbing (testing that the error detection mechanisms are still functioning correctly) before each 3-hour (maximum) flight (as a Certification Check Requirement), the probability of an error-detection mechanism failing in flight is:

$$1 - e^{(-1e\text{-}7\ *\ 3)} \approx 2.99999955e\text{-}7 \qquad (12)$$

The combined probability of an error-detection mechanism failing and an error occurring in the 4 critical bits in an hour is:

$$2.99999955e\text{-}7 * 2.8758567928056934555e\text{-}3 \approx 8.6e\text{-}10 \leq 1e\text{-}9 \qquad (13)$$

This probability is acceptable for a 3-hour flight, but increases to over 1e-^{9}/hr after just 3.5 hours of flight exposure. If the system is checked to be fault free (i.e., scrubbed) before each 3-hour flight, this procedure would probably be sufficient. However, if an error is not scrubbed from a flight, it might persist into the next flight, making the probability of an error-detection mechanism failing sometime during the life of an aircraft unacceptably high.

In general, it is not possible to build an error-detection mechanism that can be guaranteed with a sufficiently high probability to work continuously (without repair) over an aircraft lifetime. The time duration is too long for that to be achievable. Therefore, scrubbing will always be required.

There are two basic ways of scrubbing:

1. Black box (applying all possible error patterns and seeing that they are all detected)
2. White box (testing the components within the mechanism and their interconnects)

Typically, black box tests are not feasible because of the huge number of possible error patterns. Most COTS devices do not have the ability to do white box tests. This means that the error-detection mechanisms within COTS devices usually cannot be trusted and mechanisms that are allowed to be white-box tested need to be added to cover COTS devices.

6.14 OTHER ERRORS IN THE MESSAGE.

What if, as is usually done, the error detecting code covered the whole 64-byte message instead of the 4 critical bits in the example avionics data-transfer system? In this case, HD = 4 does not guarantee that all 4 altitude bits are protected. Other bits in the message may also be corrupted, using up part of the HD quota. A practical approach for handling the common case of only some subset of the bits in a message being critical is to treat the whole message as critical, if any bit is critical. For the example data-transfer system, this would mean, given the example BER of 1e-8 and 64 bytes of 8 bits each in a message, that the probability of an error in a message is:

$$1 - (1 - 1e\text{-}8)^{(64*8)} = 5.119986918e\text{-}6 \qquad (14)$$

The probability of an erroneous message within an hour is:

$$1 - (1 - 5.119986918\text{e-}6)^{72000} = 0.308326, \tag{15}$$

If this example used a perfect 16-bit random hash checksum (i.e., 1-*C* = 0.5^{16}), the probability of an undetected error in an hour would be:

$$1 - (1 - (5.119986918\text{e-}6 * 2^{-16}))^{72000} = 5.62497\text{e-}6 \tag{16}$$

If this example used a perfect 32-bit random hash checksum (i.e., 1-*C* = 0.5^{32}), the probability of an undetected error in an hour would be:

$$1 - (1 - (5.119986918\text{e-}6 * 2^{-32}))^{72000} = 8.79297\text{e-}11 \tag{17}$$

Therefore, it is likely that a good 32-bit checksum will suffice if it has good mixing, even if it is only HD = 1, as is the case for a random hash checksum. The actual coverage number for a particular checksum would need to be used to be certain; this is only an example calculation. In practice, using a cryptographically secure hash function might provide a suitable perfect random hash checksum, but analysis would be required to demonstrate that it actually achieves suitably random mixing for the intended use.

6.15 USING HD TO EXPLOIT CRCs.

Given that HD = *n* means that all 1 through *n*-1 bit errors are detected and given the above analysis for the example data-transfer system, it can be determined that a good CRC (e.g., a 10-bit CRC) would provide sufficient coverage while using fewer than 32 additional bits, as done in the previous subsection. For example, select 0 x 327 for the 10-bit CRC polynomial, which gives HD = 3 for this size of dataword.

The codeword length for a 10-bit CRC is (64 bytes * 8 bits) + 10 bits = 522 bits. Coverage at 522 bits codeword (512 bit dataword) for 0 x 327 is (23,126 undetected/(522 * 521 * 520)) possible 3-bit errors. All 1-bit and 2-bit errors are detected.

The process to determine coverage sums all probabilities where there are *k*-bit errors:

- 1 or 2 bits of error – always caught; not relevant
- 3-bit errors: (all 3-bit combinations * three bits error * rest of bits non-error)
  - combin(522,3)*$(1\text{e-}8)^3$*$(1\text{-}1\text{e-}8)^{(522-3)}$=2.35699e-17 3-bit error prob/message
  - 0 x 327 coverage: $1 - (1 - 2.35699\text{e-}17*(23126/(522*521*520)))^{72000}$, which is so small that it gives a numeric underflow when computed, but can be approximated to be 3.8543e-21 Pud/message * 72000 msgs/hr = 2.776e-16/hr undetected errors
- 4-bit errors: an even smaller probability of occurring…rounds down to zero contribution to undetected error fraction.

In other words, merely attaining HD = 3 provides sufficient coverage that the actual undetected fraction of the CRC for 3-bit errors is not material to determining if adequate error detection has been provided. Note that these computations require care with regard to numerical accuracy. Spreadsheet arithmetic often gives inaccurate answers with large exponents. A more accurate numerical tool should be used when evaluating these probabilities.

6.16 A SIMPLIFIED APPROACH TO COVERAGE.

Selecting an error-detection scheme with sufficient coverage can be simplified by using the following process. First, compute the expected number of message errors per hour for all messages with 1-bit error, then 2-bit errors, 3-bit errors, etc. This is a function of BER and total message length. At some point, the expected rate at which errors will occur is below the system target value. In other words, there will not be messages with that many (or more) bit errors at that probability (or below). For example, it may be that the probability of 3-bit errors is only 1e-12/hr and the system undetected error rate target is 1e-9/hr. Three-bit errors occur significantly less frequently than the system target rate; therefore, it does not matter if they are detected.

Next, pick a code with an HD high enough to cover as many errors as are likely. Use *k* as the number of bit errors below your target. For example, if 1-, 2-, 3-, and 4-bit error probabilities are all above 1e-9 per flight-hour, but 5-bit errors are less likely than 1e-9 per flight-hour, then *k* = 5 because *k* = 5 errors are not expected in system operation; therefore, it does not matter if they can be detected.

Finally, pick an error code that has HD = *k*. This gives perfect coverage above *k*-bit errors and conservatively assumes zero coverage at *k*. If this is on the border, more detailed coverage analysis needs to be done. In determining whether to use a checksum or CRC, the following rules can be used:

1. For *k* = 2, a checksum will suffice; the same is sometimes true for *k* = 3
2. At *k* = 4 or more, a CRC is usually needed
3. The CRC always provides better coverage at its HD than a checksum with the same HD

6.17 DATA STORAGE PROTECTION.

An alternate-usage scenario is data protection (e.g., loading executable code from flash memory, protecting data files, protecting configuration files, map information, etc.).

To the extent that the data corruption model is single bit flips, the previously described process would be used, but instead of computing so many messages per second, the math would be done based on the probability over time that the number of data bits in excess of the HD had been corrupted, weighted by the probability of detection by that number of data bits. As with network messages, a higher HD will provide significantly better coverage in a fault model because it is decreasingly likely to see more corrupt bits. In such a model, periodically checking the data ("scrubbing" the data values) will dramatically reduce the chances that multi-bit errors will accumulate.

It is also useful to look at the HD of an error-detection code over short-to-medium data word lengths to detect physically correlated bit flips (e.g., on adjacent memory array rows). This was discussed in section 5.6. A related idea, used in some systems, is to use many CRCs to protect small blocks of memory from bit errors and then a single CRC to protect the entire data array. It should be noted that such an approach is not likely to improve HD within a block, but does reduce the size of a dataword in which the HD is valid, supporting guaranteed detection of more data errors if they are not clustered in the same protected dataword. This is a variant on the idea of using multiple error codes at different dataword lengths, as was discussed in section 5.7.

If data is subject to random scrambling rather than bitwise corruption, then the size of the checksum or CRC may matter more than its mathematical properties. A *k*-bit checksum for perfectly random corruption will provide an undetected error fraction of 0.5^k (basically the same as a random hash function described). The less random the corruption, the more helpful a good mixing function is for the error-detection code. Ultimately, however, with widespread memory corruption, HD guarantees are not useful and error-detection performance in the best case degrades to 0.5^k.

If a single 32-bit CRC or checksum does not provide sufficient protection, then multiple 32-bit CRCs or checksums may provide increased protection if they are unlikely to have excessive commonly missed error patterns. Identifying such tuples of CRCs is an open question (e.g., see discussion in section 5.7) and requires careful analysis.

6.18 TRANSFER FROM NONVOLATILE TO PROGRAM RAM.

An application scenario that is a hybrid of network messages and data-storage protection is the use of a checksum or CRC to protect an image in nonvolatile memory as it is transferred to RAM. As previously mentioned, CRCs and checksums cannot provide full coverage for complex intermediate stages. At best, they can provide good coverage along the lines of a random hash approach, taking no credit for HD, but can always be vulnerable to a patterned fault of the complex intermediate stage that happens to be a pattern undetectable by the error code.

Given that caveat, transferring from nonvolatile memory to program RAM can be thought of as a three-step process. The nonvolatile memory image, the transfer process, and the RAM memory image must all be protected. All three phases may have different fault models and, therefore, different error-detection code requirements. There are too many possible combinations and interactions to give a single recommendation, but some thoughts and possible strategies follow:

1. Consider using a CRC with a relatively long HD = 6 coverage size because it will detect up to 5 bit errors at that block size even if the dataword itself is longer (see section 5.6).

2. See if it is possible to check the nonvolatile memory image before it is transmitted along a network to avoid letting transmission errors compound any faults in the original memory image.

3. Consider using a code that provides good burst-error coverage for the transmission phase.

4. If movement from nonvolatile memory to volatile memory occurs in chunks, consider protecting against errors that cause chunks to be assembled in the wrong order (e.g., faults in an address field of a data chunk being transferred).

5. Consider scrubbing all three aspects of this operation: periodically checking nonvolatile storage for faults, periodically checking the transfer mechanism for faults, and periodically checking the RAM image for faults.

7. FUTURE WORK.

A number of areas are ready for future research work.

With many systems, it is important to consider other fault models. One example is a bit slip in communication networks, for which a bit is effectively replicated or deleted from the middle of a bit stream because of timing or synchronization faults.

Interactions between physical layers and error coding are clearly issues that can compromise error-detection effectiveness (e.g., the ARINC-825 case study). Further work is needed to build upon the work shown in James, et al. [16] and to more generally ensure that protocol framing, bit encoding, scrambling, and other mechanisms do not compromise error-detection effectiveness, or at least that the compromises that take place are understood and accounted for in safety analysis. There are some physical layer attributes that will improve performance in some systems, such as the use of Manchester encoded bits where the system rejects improperly formed bit patterns [18].

Because of physical layer interaction effects, it may be that longer CRCs are desired to mitigate problems. Work on checksum and CRC effectiveness at 64 bits is minimal and should be examined as payload sizes, memory sizes, and data rates all increase.

It seems possible that cryptographically secure hash functions may be suitable for some applications in which HD = 1 is acceptable. However, further work should be done to ensure that there are no weak spots for particular hash functions in terms of providing an essentially perfect mixing function for error-detection purposes.

The use of multiple layers of error-detection codes is not well understood. For example, using multiple layers of CRCs, alternating CRCs, or combinations of checksums and CRCs is proposed frequently by application developers, but is not well understood. While there may be a way to ensure that such layering increases achieved HD, to date no such scheme has proven to be successful.

Poor understanding exists regarding interactions among compression, encryption, and error detection, especially when the error-detection code is subject to compression, encryption, or other manipulation. Additionally, though it is common to assert that encryption provides data integrity protection, it is only true if there is a way to validate the decrypted data in a reliable way. For now, HD = 1 for validating decrypted data must be assumed unless demonstrated otherwise.

If ATN-32 is being used in high-integrity applications with an expectation of better than HD = 2 performance and near the high end of the HD length limits, an analytically exact evaluation of the maximum length for higher HD lengths should be performed. The current Monte Carlo simulations provide reasonably accurate results and the shape of the curves suggests that the identified limits are close to the actual limits. In particular, simulation results conclusively demonstrate where the lower HD is present. However, there is the possibility that a lone undetected error was missed by the simulation. If simulation results are used as the basis of a safety case, at least one set of completely independent simulations should be created to validate the results in this report for that particular checksum.

Quantifying the effectiveness of Fletcher checksums and CRCs for detecting data-ordering errors is not well understood. In practice, it may be preferable to have a high-quality error-detection code for each piece of a fragmented data set that protects a sequence number.

A final, high-level opportunity for future work is exploring system-level effects on error-detection effectiveness. For example, if a control system can be constructed specifically to survive a finite number of arbitrarily corrupted messages within a limited time span, error-detection requirements may be significantly reduced on any particular corrupted message. Szilagyi and Koopman [19] show how to analyze and exploit this type of system resilience using a CAN network.

8. RESULTS.

The following is a summary of technical results and findings:

- Error-detection effectiveness depends upon multiple factors, including the size of the data to be protected, types of data errors expected, the smallest number of bit errors that are undetectable by the error code (the Hamming distance [HD]), and the undetected error fraction of errors as a function of number of bit errors present in a particular piece of data.

- Fletcher checksums provide significantly better error detection than other forms of checksums.

- The Aeronautical Telecommunication Network-32 checksum is better than a normal checksum at short message lengths, but is worse than a 32-bit Fletcher checksum at and above dataword sizes of 764 bytes.

- In many cases, cyclic redundancy codes (CRCs) provide dramatically better error-detection performance than checksums at short-to-medium dataword lengths because of a higher HD, and give somewhat better performance for long dataword lengths. The computational cost of CRCs can be reduced via a variety of techniques published in the literature.

- For many relevant aviation situations, there are far better CRCs available than the Institute of Electrical and Electronics Engineers 802.3 standard CRC-32. Also, using better CRCs can often give one or two extra bits of HD at no additional computational cost.

- In many cases, a random independent bit error fault model (bit flip faults) is the best way to evaluate CRC and checksum performance.

- Initial seed values, while potentially helpful for detecting some faults involving extra leading zeros, do not affect CRC error-detection performance.

- Checksum performance is dependent upon data values, with a data mix of 50% one bits and 50% zero bits within the data generally leading to the worst error-detection performance. The CRC error detection performance is independent of data values being processed by the CRC.

- The per-word parity bit in Aeronautical Radio, Incorporated (ARINC)-629 combined with a checksum provides HD = 4 error-detection effectiveness.

- A vulnerability to bit stuff corruption undermines ARINC-825 CRC effectiveness, degrading it from HD = 6 to HD = 2 (this is a property of the underlying commercial, off-the-shelf [COTS] Controller Area Network protocol design and not a design error in ARINC-825 per se). Additionally, the originally published (July 2011) high integrity CRC had ambiguities and typographical errors that should be corrected in an update to the standard document.

- Using CRCs and checksums in applications requires attention to numerous other factors, including: corruption of length field undermining HD, interactions between physical layer encoding (stuff bits, 8B/10B encoding, data scramblers) and error-detection codes, interaction between encryption and error-detection codes, and correlated data errors caused by data layout patterns (e.g., adjacent rows in a memory array). Error-detection mechanisms and the data they protect must be periodically scrubbed to avoid an accumulation of latent faults. Additionally, complex circuits may introduce correlated bit errors or otherwise undermine error-detection effectiveness. Each of the factors listed here have the potential for invalidating any arguments/calculations used to determine the effectiveness of CRCs and checksums.

- Using multiple error-detection codes in conjunction can help, but not always to the degree hoped. In general, applying multiple error-detection codes to the same dataword is not likely to increase HD. However, nesting error-detection codes (many small datawords, each individually protected, plus an overarching error-detection code) can be useful and may even result in increased HD for the composite approach.

- There is no one-size-fits-all answer as to which error-detection code to use. The following steps are recommended for determining if a particular error code meets integrity requirements:

  – Determine which functions must be considered.

  – Determine the maximum acceptable probability of failure for each function.

  - Determine exposure to failures.
  - Calculate maximum allowed failure probability.
  - Determine error model.
  - Determine which errors can be dangerous.
  - Calculate dangerous error probabilities.
  - Calculate required error coding coverage and whether the proposed error code attains that coverage.

- In determining if undetected error probabilities are low enough, an error code should have a sufficiently large HD so all errors that are expected to occur within a system's operating life are guaranteed to be detected (as long as a random independent fault model accurately reflects the errors that will be seen in operation).

- When deploying an error-detection code, such assumptions as bit error ratio (BER) should be ensured to be true not only at design time (e.g., ensure BER in avionics environment is acceptable when using COTS network hardware created for desktop computing), but also during operation (e.g., ensure BER has not degraded as a result of equipment aging). Establishing an accurate error model is critical in determining the probability of error-detection escapes.

Informally, the "Seven Deadly Sins" (i.e., bad ideas) of CRC and checksum use are:

1. Picking a CRC based on popularity instead of analysis.
2. Saying that a good checksum is just as good as a poorly chosen CRC.
3. Evaluating randomly corrupted data detection capability when a BER model is more appropriate.
4. Blindly using factorization or other attributes to pick a CRC instead of analytically exhaustive analysis.
5. Failing to protect a message length field.
6. Failing to pick an accurate fault model and apply it (treating a checksum as "body armor" for a safety argument).
7. Ignoring interactions between error codes and symbol encoding (including stuffing and scrambling).

9. REFERENCES.

1. Prange, E., "Cyclic Error-Correcting Codes in Two Symbols," AFCRC-TN-57-103, ASTIA Document No. AD133749, Air Force Cambridge Research Center, 1957.

2. Koopman, P. and Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," *Proceedings of International Conference on Dependable Systems and Networks*, Florence, Italy, June 2004, pp. 145–154.

3. Ray, J. and Koopman, P., "Efficient High Hamming Distance CRCs for Embedded Networks," *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, Washington, DC, USA,IEEE Computer Society, 2006, pp. 3–12.

4. Matsumoto, M. and Nishimura, T., "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, 1998, pp. 3–30.

5. Maxino, T. and Koopman, P., "The Effectiveness of Checksums for Embedded Control Networks," *IEEE Transactions on Dependable and Secure Computing*, Vol. 6, No. 1, January–March 2009, pp. 59–72.

6. Koopman, P., "32-Bit Cyclic Redundancy Codes for Internet Applications," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA, *IEEE Computer Society*, 2002, pp. 459–472.

7. Usas, A.M., "Checksum Versus Residue Codes for Multiple Error Detection," *Proceedings of Eighth Annual International Symposium on Fault-Tolerant Computing*, 1978, pp. 224.

8. Fletcher, J.G., "An Arithmetic Checksum for Serial Transmissions," *IEEE Transactions on Communication*, Vol. 30, No. 1, January 1982, pp. 247–252.

9. Nakassis, A., "Fletcher's Error Detection Algorithm: How to Implement It Efficiently and How to Avoid the Most Common Pitfalls," *Computer Communication Review*, Vol. 18, No. 5, October 1988, pp. 63–88.

10. Deutsch, P. and Gailly, J.L., "ZLIB Compressed Data Format Specification Version 3.3," IETF RFC 1950, May 1996.

11. "Manual on Detailed Technical Specifications for the Aeronautical Telecommunication Network (ATN) Using ISO/OSI Standards and Protocols, Part 1–Air-Ground Applications," Doc 9880, International Civil Aviation Organization, Montreal, Canada. First Edition, 2010.

12. Tran, E., "Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol," M.S. Thesis, CMU ECE, May 1999.

13. “General Standardization of CAN (Controller Area Network) Bus Protocol for Airborne Use,” ARINC Specification 825-2, Aeronautical Radio Inc., Annapolis, July 14, 2011.

14. Driscoll, K., Hall, B., Koopman, P., Ray, J., and DeWalt, M., “Data Network Evaluation Criteria Handbook,” FAA report DOT/FAA/AR-09/24, June 2009.

15. “FlexRay Communications System Protocol Specification Version 3.0.1,” *FlexRay Consortium*, 2010.

16. James, L., Moore, A., Glick, M., and Bulpin, J., “Physical Layer Impact Upon Packet Errors,” *Proceedings of the Passive and Active Measurement Workshop,* April 2006.

17. FAA Advisory Circular 20-156, Aviation Databus Assurance, August 2, 2006.

18. Tzou Chen, S.C. and Fang, G.S., “A Closed-Form Expression for the Probability of Checksum Violation,” *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 10, No. 7, July 1980, pp. 407–410.

19. Szilagyi, C. and Koopman, P., “A Flexible Approach to Embedded Network Authentication,” DSN 2009, pp. 165–174.

10. GLOSSARY.

^: Exponentiation symbol for CRC polynomials (e.g., *x*^4 means “*x* to the 4th power”)

802.3 Polynomial: The CRC polynomial defined by the IEEE 802.3 Ethernet standard.

8B/10B Encoding: A mapping of groups of 8 bits of data into 10-bit symbols.

Adler Checksum: A variant of the Fletcher checksum that uses the largest prime number smaller than the power of two corresponding to the chunk size as the addition radix.

Bit Error Rate: See bit error ratio (equivalent term in common use; ratio is preferred over rate because this term, as calculated and used, is dimensionless, whereas a rate should have a dimension of frequency or inverse time).

Bit Error Ratio (BER): The ratio of bits in a codeword that are erroneous compared to total number of bits. For example, a BER of 1e-5 (alternately written as 10^{-5}) means that 1 bit in 100,000 is erroneous. The BER implies a random independent bisymmetric bit errors.

Bit Flip: Another term for bisymmetric bit error in which an error inverts a single bit value.

Bisymmetric Bit Error: A fault model for which erroneous bits have inverted values (i.e., a 1 changes to a 0, or a 0 changes to a 1).

Burst Error: A set of bit errors contained within a span shorter than the length of the codeword. The burst length is the distance between the first and last bit errors. Zero or more of the intervening bits may be erroneous. Most error codes discussed in this report have special capabilities for detecting short burst errors. Usually, a code that detects a certain length of burst error will also detect all shorter burst errors.

Byte Parity: Use of one parity bit per byte of data, as is commonly done for serial data transmissions.

Check Sequence: An error-detecting code value stored with or transmitted with associated data. A dataword plus a check sequence makes up a codeword.

Checksum: An error-coding technique that primarily involves adding chunks of data together to yield a check sequence. Some checksums use a tuple of concurrent running sums. This term is also used as a synonym for the check sequence produced by this technique.

Chunk: A tuple of bits from a dataword used to compute a CRC or checksum. This may or may not be the same as the word size, and may or may not be the same as the check sequence size. For example, a 16-bit Fletcher checksum would have a chunk size of 8 bits, meaning that the data is processed 8 bits at a time, resulting in the update of a pair of 8-bit running sums, ultimately forming a 16-bit check sequence value.

Code Size: The number of bits in the error code computation result, which by construction is the same as the FCS size.

Codeword: A dataword combined with a check sequence.

Cyclic Redundancy Check: See Cyclic Redundancy Code (equivalent term in common use).

Cyclic Redundancy Code (CRC): An error-coding technique based upon polynomial division of a dataword by a feedback polynomial, yielding the remainder of the division as a check sequence. This term is also used as a synonym for the check sequence produced by this technique.

Dangerous: Means, for the purposes of this report, data errors that could lead to unsafe system behaviors.

Dataword: The data value being protected by a check sequence. A dataword can be any number of bits and is independent of the machine word size (e.g., a dataword could be 64 Megabytes).

Detectable Error: A corrupted codeword that can be detected as having been corrupted by an error check.

Divisible by (x + 1): A CRC feedback polynomial that is evenly divisible by the term (x + 1), providing detection of all odd numbers of bit errors.

Error Check: The act of comparing a check sequence with the result of an error code computation to detect potential data corruption.

Error Code: An algorithm or computation performed on a dataword that yields a check sequence for the purpose of error detection or error correction.

Exclusive OR (XOR): A logical operation that is true when an odd number of its inputs are 1 and false when an even number of its inputs are 1. For a 2-bit input, the output value is 1 for inputs of 0,1 or 1,0, and the output value is 0 for inputs of 0,0 or 1,1.

Factorization: See Polynomial Factorization.

Feedback Polynomial: A polynomial used as the divisor by a CRC. In a CRC implementation, this corresponds to the position of "1" bits in the value XORed into the running computational value of the CRC algorithm.

Fieldbus: In this report, a generic term for an embedded communication network used for industrial control.

Fletcher Checksum: A checksum that is performed on a pair of running sums using one's complement addition.

Frame Check Sequence (FCS): A check sequence that resides in a network message frame. In this report, it is also used generically to refer to the error-detection bits within a codeword, even if they are in memory rather than in a network message.

Hamming Distance (HD): The minimum number of bits that differ between any two valid codewords. This number is the minimum number of bit flips sufficient to create an undetectable error (transforming one codeword into another corrupted, but valid, codeword).

Hamming Weight (HW): The number of undetectable errors involving a particular number of bit errors for a given error code and dataword size. A HW of zero means all possible errors are detected for that particular number of errors. A given error code and dataword length have a tuple of HW values, with one HW for each possible number of bit errors.

Internet Checksum: A 16-bit one's complement addition checksum used to protect the header of an Internet packet.

Irreducible Polynomial: A polynomial that is evenly divisible (via polynomial division) by only itself and 1. This is somewhat analogous to an integer prime number; however, the concept of a primitive polynomial (included within this same list) has some mathematical properties more similar to the properties of an integer prime number.

Linear Feedback Shift Register (LFSR): A pseudo-random number generator based on the same math as a CRC. An LFSR is said to be maximal length if it generates all possible values (except zero) before repeating a value. For an N-bit LFSR (internal state has N bits), the maximal generated length is $2^N - 1$.

Longitudinal Redundancy Check (LRC): A parity-based error code created by computing bit-by-bit parity across a dataword (accomplished via XORing all the data chunks of a dataword). The LRC is often called a checksum, although the operation is only a sum on a bit-by-bit basis, rather than being a true integer sum.

Manchester Encoding: A return-to-zero (RZ) bit encoding strategy for encoding data bits for which each 0 or 1 symbol is a pair of alternating value physical transmission bits (either {0,1} or {1,0}). This results in transmitting twice as many physical bit values as data bits, providing significant ability to detect errors that affect only one of a pair of physical bit values.

Masquerade Fault: A message sent by an untrusted node or a message corrupted in transit that has header information changed so that it appears to come from some other, trusted node (the untrusted transmitter in effect masquerades as a trusted transmitter).

Monte Carlo Simulation: A repeated set of stochastic simulations. In this report, each simulation trial randomly selects a set of bit error locations and dataword bit values to determine if that particular bit pattern is detectable for that particular dataword, using a particular error detection code.

Non-Return-to-Zero (NRZ): A bit encoding for which the signal is not required to return to zero for a portion of each bit time (as opposed to RZ encoding)

One's Complement Checksum: An error code based on adding all the data chunks of a dataword using one's complement addition (for 8 bits, this is modulo 255 instead of modulo 256). To convert the two's complement addition used by most modern computers into one's complement addition, any carryout of the twos complement sum is added back into the sum's least significant bit.

Parity: An error-coding technique based upon XORing data bits to yield a check sequence. A parity bit is a single-bit result of computing parity across an entire dataword. Parity is usually computed only on relatively small datawords.

Polynomial: A polynomial representation of a binary value for which each bit position is used as the coefficient of a power of *x* corresponding to the bit position. For example, binary value 1011 can be represented as 1**x*^3 + 0**x*^2 + 1**x*^1 + 1**x*^0 = *x*^3 + *x* + 1.

Polynomial Factorization: The irreducible factors of a polynomial, analogous to the prime factorization of an integer. The factorization of a polynomial can partially—but not completely—predict its effectiveness for small numbers of bit errors when used in a CRC.

Primitive Polynomial: A polynomial that is irreducible and has an additional property that allows it to divide the polynomial *x*^*D* – 1 evenly for no positive integer *D* smaller than $2^N - 1$. A primitive polynomial produces the maximal length output sequence of $2^N - 1$ bits when used in an LFSR.

Probability of Undetected Error (Pud): The probability that an error under certain conditions will happen to be undetectable. This typically assumes a random independent BER for which a small number of errors per codeword are much more likely than a large number of errors per codeword.

Return-to-Zero (RZ): A bit encoding in which the signal returns to zero for a portion of each bit time (as opposed to NRZ encoding) in which zero is a signal level midway between the levels used to represent a binary one or a binary zero.

Seed Value: The value placed in a checksum or CRC accumulation register before the start of the error-code computation. Using a nonzero seed value increases error-detection capability for datawords with an incorrect length that have initial data chunk values of zero and may help protect against masquerade faults if different data sources use different seeds. If no seed value is stated, the seed value is assumed to be zero.

Tuple: An ordered set of values.

Two's Complement Checksum: An error code based on adding all the data chunks of a dataword using two's complement addition. Most modern computers use two's complement addition.

Undetected Error: A situation in which an error check fails to detect an error in a codeword because of a data corruption of a pattern that cannot be detected by the error code being used. In practice, this means that a correct codeword has been mutated to a different, valid (but incorrect) codeword, and therefore has been subjected to at least as many bit errors as the HD.

Weight: The number of 1 bits in a data value. An odd weight means an odd number of 1 bits (1, 3, 5, 7, etc.). An even weight means an even number of 1 bits (0, 2, 4, 6, etc.). For example, a CRC polynomial divisible by (x + 1) detects all odd weight error patterns. Not to be confused with HW, which is a different use of the word.

Word Parity: Use of one parity bit per hardware word of data, as is done in some networks. Note that in this case the "word" is a hardware word (perhaps 16 or 32 bits) and not necessarily a dataword for a checksum or CRC that may be used in addition to the parity bit.

APPENDIX A—LITERATURE REVIEW AND DISCUSSION

A-1. THE CYCLIC REDUNDANCY CODE AND CHECKSUM.

This appendix documents the literature review performed at the beginning of the task.

A-1.1 Checksum Study Comparative.

Reference: Maxino, T. and Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," *IEEE Transactions on Dependable and Secure Computing*, January–March 2009, pp. 59–72 [A-1; 5 in the main document].

This paper compares various types of parity, checksum, and cyclic redundancy code (CRC) computations using probability of undetected error (Pud) and Hamming distance (HD) as performance metrics at various codeword lengths. Quantitative comparisons among alternative error-detection schemes are provided. Some of the conclusions are summarized as follows:

- A one's complement addition (one which does not enforce a conversion from all ones to all zeros in the check sequence values) is always superior to a two's complement addition due to capturing the effects of carrying out the addition operation. Either method of addition is significantly superior to parity schemes, such as longitudinal redundancy check (LRC).

- Fletcher checksums are almost always better than Adler checksums. In the small regions in which Adler checksums are better, they do not provide significant error-detection improvements and are more expensive to compute than Fletcher checksums. Therefore, Fletcher checksums are preferred to Adler checksums for general purpose use (assuming that the Fletcher and Adler checksum sizes are the same).

- A well-chosen CRC is significantly better than any checksum approach. For example, an optimal 12-bit CRC provides better error detection than a 16-bit Fletcher checksum.

A-1.2 Other Checksum Performance Studies.

Other reports that reference A-1 builds upon investigate checksum performance. These include references A-2 through A-4.

Reference A-5 provides a comparison of extended-precision checksums against smaller CRCs for the restricted purpose of checking program image integrity.

A-2. GENERAL CRC RESEARCH.

A-2.1 The CRC Implementation.

Prange introduced the concept of a CRC in 1957 [A-6; 1 in main document]. Briefly, a CRC works by performing polynomial division of a dataword by a generator polynomial over Galois Field (2). The remainder of that division provides a check sequence. In practice, shift-and-exclusive OR (XOR) and table lookup techniques exist that permit the polynomial division to map efficiently onto computer hardware and software. Peterson and Brown provide a more thorough early treatment of the subject [A-7].

Feldmeier provides a list of techniques for fast implementation of checksums and CRCs as well as a computational speed comparison [A-8]. A 64-bit dataword size was used and speed comparisons were performed on a SPARC CPU, which may be more representative of desktop processors than embedded processors.

Ray and Koopman discuss fast CRC computation techniques and provide CRC polynomials that were chosen specifically to support fast software computation [A-9; 3 in main document]. Evaluation of polynomials is given based on HD, minimum Hamming weight (HW) at the first non-zero HD (which predicts Pud for low bit error rates), and polynomial structures suitable for high-speed software implementation.

Braun and Waldvogel provide a technique for making incremental changes to CRCs when a small portion of a message changes [A-10].

A-2.2 The CRC Performance Analysis.

Wheal first discusses the use of a Pud model for evaluating CRCs [A-11].

Fujiwara et al. provide a performance analysis for shortened Hamming codes (not CRCs), setting the pattern for much early CRC performance analysis work [A-12]. The Pud is shown for some codes for different dataword lengths for bit error ratios (BERs) values or 0.5 to 1e-7. A summary comparison is then presented that looks for the peak undetected error rate for a BER of up to 0.5, which is an extremely high BER for an embedded network, but is representative of the mathematical worst case for a coding scheme.

In 1989, Fujiwara et al. [A-13] determined the HD values of the Institute of Electrical and Electronic Engineers (IEEE) 802.3 standard polynomial [A-14] for codeword sizes up to the maximum Ethernet transmission unit size of 12,144 bits. The evaluation was performed both in terms of HD and Pud for varying BER values.

Costello et al. provide an overview of error-control coding and how CRCs and checksums fit into the overall field [A-15]. The brief section on CRCs suggests picking polynomials that are primitive multiplied by a factor of $(x + 1)$ to ensure all odd-weight error patterns are detectable. This report proposes measuring CRC effectiveness as a percentage of all detectable error patterns (although not stated in this report, the result would be the same for all CRC polynomials). Checksums are presented as an alternative with the same effectiveness by that metric. It is noted that both a CRC and checksum $(1 – 2^k)$ catch errors for a *k*-bit check sequence.

Sheinwald et al. [A-16] consider the use of a CRC [A-17] compared to Fletcher and Adler checksums. They conclude for a certain set of assumptions that a CRC has 12,000 times better error-detection capability than a Fletcher checksum and 22,000 times better error-detection capability than an Adler checksum. They also compare CRCs and checksums according to the metrics of instructions per dataword byte to compute, sensitivity to data values (checksum Pud is sensitive to data values; CRC Pud is not), and lookup table size. Comparisons were made for both a low-noise channel and burst errors.

A-2.3 Good CRC Polynomials.

Castagnoli et al. published optimal 16-bit CRCs [A-18] based on an exhaustive search. They identified the importance of using different CRCs, depending upon the dataword length of interest. Evaluation of CRC effectiveness was based on Pud, with HD also mentioned.

Castagnoli et al. published effective 24-bit and 32-bit CRCs [A-18] based on a partial search of possibilities using special-purpose search hardware. The search was limited to less than all possible polynomials because there was limited computational power. Newly proposed polynomials were compared to existing standard polynomials based on HD and Pud.

Chun and Wolf also created special-purpose search hardware for evaluating CRCs [A-19]. The evaluation criteria were minimum Pud for BER values of approximately 0.01–0.5.

Funk published a study of the best CRCs and other shortened linear codes [A-20]. His results included an exhaustive search of all codes with a check sequence size of 5–16 bits. In a few cases, a *k*-1-bit CRC plus a parity bit slightly outperformed a *k*-bit CRC. Evaluation of performance was done based on HD and Pud for BER values starting at 10^-4 and higher.

A paper published by Baicheva et al. provided a suggestion for good 8-bit CRCs [A-21]. Only some polynomials with particular factorizations were considered. The criteria used for evaluating polynomials were: dataword lengths for which the codes are "monotonous," "good," and "proper," where these are specific mathematical properties. Additionally, minimum distances (HD) and covering radius are given for each code. Finally, code weight (number of bits in the polynomial) is considered as a factor involving expense of computation. Ultimately, CRCs were evaluated for Pud at a relatively high BER or 0.01–0.5 for a single dataword length of 40 bits (at such high bit error rates, an aviation network would be expected to be inoperative because of essentially receiving no fault-free messages). These evaluation criteria are mathematically based, such as the definition of "proper," which is based on how a CRC behaves as the BER approaches 50% [A-22].

Baicheva et al. subsequently published a paper identifying good 16-bit CRCs [A-23] according to the same evaluation criteria as Baicheva, et al. [A-21]. The 16-bit CRC polynomials that were best for each dataword length were identified. Only some polynomials with particular factorizations were considered.

Kazakov also published a study of CRC polynomials, using minimum HWs as the evaluation criterion [A-24]. The results include a polynomial that is optimal for each dataword length from 255 bits to 32,768 bits.

Koopman published an exhaustive search of all possible 32-bit CRC values to find the polynomial that provided HD = 6 at the longest possible value [A-25; 6 in main document]. The polynomial identified was not in the search space typically addressed based on promising polynomial factorizations, leading to further work with exhaustive searches in later papers. The evaluation criterion for this search was based on HD.

Koopman and Chakravarty later published a report that gave optimal CRC polynomials for sizes of 4–16 bits. The evaluation criteria were based on HD and lowest HW at that HD. An additional criterion was identifying a polynomial that also had a better HD at a shorter dataword length. This was to improve error-detection performance for CRCs used for network messages, which in embedded systems can consist of many short messages (which would then achieve high HD) and fewer long messages (which would get a lower, but still optimal, HD).

A-3. AVIATION APPLICATIONS.

A-3.1 Aeronautical Telecommunication Network Checksum.

Reference: "Manual on Detailed Technical Specifications for the Aeronautical Telecommunication Network (ATN) Using ISO/OSI Standards and Protocols, Part 1–Air-Ground Applications," Doc 9880, International Civil Aviation Organization, Montreal, Canada. First Edition, 2010 [A-26; 11 in main document].

Chapter 6 of this document identifies a default ATN message checksum for air/ground uplink and downlink data transfers. This is a variant on the Fletcher checksum that uses four 1-byte cascaded sums to produce a 32-bit checksum (a typical Fletcher checksum operates in a similar manner, but uses only two such sums). After the computation of the running sums, the 32-bit checksum value is computed using various combinations of the running sum values defined in the standard. The sum results are scrubbed so that all hexadecimal values of 0xFF are forced to 0x00 for one's complement additions.

A-3.2 Aeronautical Radio, Incorporated-629.

Reference: "Multi-Transmitter Data Bus Part 1 Technical Description," ARINC Specification 629P1-5, *Airlines Electronic Engineering Committee*, Annapolis, March 31, 1999 [A-27].

This aviation databus standard uses checksum and CRC packet error detection. Section 2.6.2 states that a checksum can be used for "small, simple data packets, with high integrity requirements" whereas a CRC should be used for long and complex data packets.

Section 5.3.7.1(a) defines a checksum as a 16-bit "sum of all words." Given a lack of qualifying language, this appears to be a two's complement checksum.

Section 5.3.7.1(b) defines a 16-bit CRC with polynomial $P(x) = x$^16 + x^12 + x^5 + 1 with a seed value of all 1s.

The CRC is computed across the entire message, including the label word. No separate CRC is used for the word count field in the header. The maximum transmission length is 257 words, including CRC (level 3 bus access, section 2.1.2).

A-3.3 Time-Triggered Protocol Version C.

Reference: Haidinger, W., “Analysis of the CRC Polynomial Used in TTP/C,” Research Report 11/2003, Real-Time Systems Group, Technical University of Vienna, Feb 19, 2003 [A-28].

This protocol was originally developed for drive-by-wire automotive applications, but is being applied in small aircraft flight controls.

This report verifies that the Time-Triggered Protocol Version C (TTP/C) [A-29] data protection polynomial is HD = 6 up to 2016 bits of payload and HD = 4 up to the maximum message length, and documents the analysis methods. Header field protection is not needed because each message is deterministically placed, with header information (including expected message length) stored in a database rather than being sent in a message.

A-3.4 Ethernet-Based Standards.

Some aviation standards are built on Ethernet, using the standard IEEE 802.3 CRC and Internet checksum. While additional measures are taken to ensure data integrity, from a CRC and checksum point of view, they are no different than IT networks unless additional measures are taken.

Boeing is said to use an additional safety CRC in its end-to-end data protection approach for Avionics Full-Duplex Switched Ethernet [A-30], but details are not publicly available.

A-3.5 The IEEE 1394 Firewire.

Reference: IEEE Std 1394-1995, “IEEE Standard for a High Performance Serial Bus,” August 1996, ISBN 0-7381-1203-8 [A-31]. (Firewire is also used as the basis for aviation standards SAE, Inc. (SAE) AS-1A-3 and MIL-1394.)

The IEEE Std 1394-1995 requires a 32-bit header CRC that protects data length and a 32-bit data CRC at the end of any packet that has a data payload in addition to the header. The 802.3 polynomial is used.

A newer version of the standard, IEEE Std 1394.3-2003, requires the use of a 32-bit CRC for configuration read only memory and for directories.

The bit order of CRC computation is the reverse of the bit order of transmission, reducing burst-error-detection coverage of the Firewire CRC below the normal 32 bits [A-9].

A-3.6 Federal Aviation Administration Digital Systems Validation Handbook.

Reference: *Handbook – Volume II, Digital Systems Validation*, Chapter 18, “Avionic Data Bus Integration Technology,” DOT/FAA/CT-88/10, FAA Technical Center, Atlantic City NJ, November 1993 [A-32].

This handbook describes the characteristics of a number of aviation network protocols and lists protocols deployed on a variety of aircraft. It includes discussion of checksum and CRC approaches.

Based on the information in this handbook Aeronautical Radio, Incorporated (ARINC)-429 uses one parity bit per 32-bit dataword. A 16-bit CRC is used to verify groups of datawords or data strings (section 5.1.3.2.1.1). The Comité Consultatif International Téléphonique et Télégraphique (CCITT) 16-bit polynomial is used: x^16 + x^12 + x^5 + 1.

Other protocols use per-byte parity and 8-bit or 16-bit CRCs/checksums, although specifics are not generally given beyond the use of the ARINC-429 CRC polynomial.

A-3.7 Federal Aviation Administration Order 8110.49.

Reference: "Software Approval Guidelines Order 8110.49 Chg 1," Federal Aviation Administration, 9/28/2011 [A-33].

The relevant sections of this document discuss software conformity inspection for aviation applications.

Section 4-4-b-1 gives checking for a correct CRC value as an example approach for ensuring that the correct software part number and version of a system has been loaded.

Section 5-2f indicates that a CRC can also be used for a data integrity check for software corruption when installed, and chapter 5-4b indicates it is also useful to do this before and after maintenance. No specifics as to which CRC should be used are given.

A-3.8 Ultra-Dependable Systems.

Reference: Paulitsch, M., Morris, J., Hall, B., et al., "Coverage and Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," *Proceedings of the International Conference on Dependable Systems and Networks,* June 2005, pp. 346-355 [A-34].

This paper specifically addresses potential issues in using CRCs for aviation and similar ultra-dependable applications. Similar issues would apply to checksum use. A set of rules for appropriate CRC use is given.

Potential issues with typical evaluation criteria include varying BER and susceptibility to Byzantine errors.

A potential issue with applications is using a CRC to protect implicit datawords that have significant changes between transmitted values. This applies to group membership vectors used in some modes of TTP/C operation, but could also apply to seed values used for masquerade detection.

Another potential application issue is an intermediate communication stage corrupting a message and the accompanying CRC in a systematic manner that leads to undetected errors.

A-4. AUTOMOTIVE APPLICATIONS.

A-4.1 ISO-26262.

Reference: International Standard ISO 26262-5:2011(E), “Road vehicles – Functional Safety – Part 5: Product Development at the Hardware Level,” First Edition, International Standards Organization, Geneva, Switzerland, November 15, 2011 [A-35].

This is an automotive safety standard that is intended to encompass safety-critical drive-by-wire systems. The 8-bit and 16-bit CRCs are proposed as an error-detection mechanism for data values in random access memory (RAM), network messages, and flash memory.

Appendix D of part 5 of ISO-26262-5 provides an extensive list of techniques for error detection, including discussion of parity, checksums, and CRCs.

This appendix rates a variety of techniques in terms of diagnostic coverage. Section D.2.7.6 gives the following guidelines for diagnostic coverage as well as some example CRC polynomials to achieve these levels:

- Low coverage: CRC HD = 2 or less
- Medium coverage: CRC HD = 3 or more
- High coverage: Can be reached with a CRC for data and ID corruption, but no specific HD is provided

The example CRCs are taken from Koopman and Chakravarty [A-36; 2 in main document]. Checksum coverage is considered low for network messages because of the low HD achievable, making CRCs the preferred error-detection approach for critical data.

The ISO-26262-5 Section D.2.4.3 Note 2 states: “Use of an 8-bit CRC is not generally considered state of the art for memory sizes above 4k” (in context, this means 4 kilobytes.)

A-4.2 FlexRay.

Reference: “FlexRay Communications System Protocol Specification Version 3.0.1,” *FlexRay Consortium*, 2010 [A-37; 15 in main document].

This is a safety-critical network protocol created for automobiles, with applications such as drive-by-wire specifically in mind. FlexRay and TTP/C are considered state-of-the-art for automotive applications, with FlexRay having a dominant position in automotive drive-by-wire designs. Based on these standards, HD = 6 is considered best practice for safety-critical network message CRC error detection.

A fixed-length header includes an 11-bit header CRC that protects the FrameID, payload length, and other important status bits, covering 20 bits of header dataword (see section 4.2.8 of the main document). It uses polynomial *x*^11 + *x*^9 + *x*^8 + *x*^7 + *x*^2 + 1 to provide HD = 6 protection for the header. This is specifically intended to protect against corruption of the length field, which could cause a receiver to look for the CRC in the wrong place, undermining the achieved HD.

The payload segment contains 0–254 bytes of data. It uses a CRC that provides HD = 6 up to 248 bytes and HD = 4 above that. It uses polynomial: *x*^24 + *x*^22 + *x*^20 + *x*^19 + *x*^18 + *x*^16 + *x*^14 + *x*^13 + *x*^11 + *x*^10 + *x*^8 + *x*^7 + *x*^6 + *x*^3 + *x* +1.

The two redundant channels use different seed values. Channel A uses 0xFEDCBA, whereas Channel B uses 0xABCDEF.

A-4.3 AUTOSAR End-to-End Protection.

Reference: "Specification of SW-C End-to-End Communication Protection Library," V 1.1.0, R4.0 Rev 2, AUTOSAR Standard, 12/17/2009 [A-38].

This specification provides different profiles for different levels of integrity assurance for automotive communication integrity. The CRCs are added to data payloads to provide error detection beyond detection provided by underlying network protocols. Section 7.3.4 discusses CRC calculations.

Profile 1 uses an 8-bit CRC from the SAE J1850 standard (*x*^8 + *x*^4 + *x*^3 + *x*^2 + 1) and a seed value of 0xFF: The result of the CRC computation is XORed with 0xFF. The 8-bit CRC is placed in the first byte of the message rather than the last byte.

Profile 2 uses a different 8-bit CRC (*x*^8 + *x*^5 + *x*^3 + *x*^2 + *x* + 1) for a maximum codeword length of 256 bytes. This polynomial provides HD = 2 at that maximum length.

A-4.4 BMW Automotive Study.

Reference: Mehrnoush, R., Muller-Rathgeber, B., and Steinbach, E, "Error Detection Capabilities of Automotive Network Technologies and Ethernet – A Comparative Study," *Proceedings of the IEEE Intelligent Vehicles Symposium*, Istanbul, Turkey, June 2007, pp. 674–679[A-39].

This study concludes that plain Ethernet is not as effective as FlexRay and Controller Area Network [A-40] for a range of random independent bit errors and recommends augmented error detection if Ethernet is used because of economic or other considerations (the conclusions in this paper state that Ethernet can provide the same data-protection capabilities if modified or augmented, but does not present analysis to support this conclusion).

A-5. INDUSTRIAL CONTROL AND RAIL APPLICATIONS.

A-5.1 Open-Safety.

Reference: "OpenSAFETY: The First Open and Bus-Independent Safety Standard for All Industrial Ethernet Solutions," available at http://open-safety.org/ (accessed January 25, 2012) [A-41].

Materials downloaded from: "Software Package for the Development of Safety Nodes (SN) and Safety Configuration Manager (SCM), http://www.ixxat.com/ethernet_powerlink_safety_stack_en.html (accessed on January 25, 2012) [A-42].

This is a high-level safety layer that is said to be portable to any fieldbus. A wide variety of implementations exist, including Ethernet/Internet Protocol (IP). It is said to be International Electrotechnical Commission (IEC) 61508 Safety Integrity Level (SIL) 3 (which means it is suitable for a certain level of safety-critical operations). The organization offers a reference implementation with source code implementation and software interface reference manual available. A detailed protocol specification does not appear to be publicly available.

Open-Safety uses a 32-bit CRC to verify source code file integrity for the unit test suite. Developers are told they must pass those unit tests when including the functions in their applications. Thus, the CRC is being used as an integrity check on a protocol implementation compliance test suite.

The CRCs are used to protect parameters stored in memory and packets. Two CRCs are used: an 8-bit CRC with unspecified polynomial for datawords of 0–8 bytes and a 16-bit CRC with unspecified polynomial for datawords of 9–254 bytes.

The reference implementation provided by the organization has CRC computations that always return all 1 values. (0xFF for 8-bit CRC and 0xFFFF for 16-bit CRC.) A query to the distributor of the reference implementation asking for clarification of the situation was unanswered.

A-5.2 Process Field Bus Safety Profile.

Reference: "Profibus–DP/PA: Profisafe–Profile for Failsafe Technology, v 1.0, Document No. 740257," March 1999, p. 33[A-43].

Process field bus safety profile (PROFIsafe) is a usage profile for the process field bus (PROFIBUS) industrial automation network protocol. It uses CRCs in multiple ways.

A CRC is used to check the integrity of variables in memory (process data) with the size of the CRC and time period varying, depending upon the size of the process data and the SIL (see page 23 of reference A-43.

A 16-bit or 32-bit CRC is used to guard against undetected data corruption in network messages (section 4.3.5 of reference A-43. Masquerading is prevented by using seed values for CRCs (called CRC keys in this protocol).

A-5.3 Nuclear Regulatory Commission Publication/CR-6463.

Reference: "Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems," SoHar Incorporated, U.S. Nuclear Regulatory Commission, Washington DC, June 1996 [A-44].

This document provides guidelines for software language use in nuclear power plant safety systems on a language-by-language basis. It generically discusses checksum best practices (the word "checksum" seems to be used in a generic sense that encompasses CRCs).

Section 4.1.1.7 states that a checksum of the program image should be checked at power-up to detect program corruption. Section 5.2.2.1 indicates that it is good practice for a system to protect RAM values with a checksum and shut down if there is a checksum error.

A-5.4 Nuclear Regulatory Commission Publication/CR-6082.

Reference: "Data Communications," Lawrence Livermore National Laboratory, US Nuclear Regulatory Commission, Washington DC, August 1993 [A-45].

This document provides regulatory guidance for data communications in nuclear power plant safety applications. It provides lists of questions to be asked when evaluating critical data networks, but does not have significant depth on the topic of checksums and CRCs. It is noted in section 2.5.3 that CRC use is a common technique, but no particular CRC polynomial or size is recommended.

A-5.5 Train Control Network.

Reference: Koopman, P. and Chakravarty, T., "Analysis of the Train Communication Network Protocol Error Detection Capabilities," Carnegie Mellon University Technical Report, February 25, 2001 [A-46; 18 in main document].

The Train Control Network is a pair of protocols used for within-car and whole-train rail communications.

The multi-function vehicle bus within-car protocol uses a 7-bit CRC plus parity bit to create an 8-bit checksum. When combined with Manchester encoding, this combination is predicted to yield no more than 10^-6 undetected errors per year for a single continuously operating system.

The wire train bus whole-train protocol uses the high level data link control (HDLC) protocol with the addition of Manchester encoding. This uses the standard CCITT 16-bit CRC polynomial.

A-5.6 VTT Study.

Reference: Alanen, J., Hietikko, M., and Malm, T., “Safety of Digital Communications in Machines,” VTT Research Notes 2265, VTT Industrial Systems, 2004 [A-47].

This document discusses industrial control networks for machine controls (e.g., construction equipment), including checksum and CRC approaches. A summary of findings is as follows (in many cases, primary source documents are not available, so the statements that follow are taken at face value from this source):

- Section 2.2: Berufsgenossenschaftlichen Instituts fur Arbeitssicherheit Guidelines include the use of an additional CRC in critical data and do not take credit for any CRC provided by the underlying network infrastructure.

- Section 2.2.1: The EN 50159-1 is a rail application standard that suggests using a single pessimistic equation for residual error property of 2^-*c* (where *c* is the length of the safety code in bits). In other words, it suggests taking no credit for HD, but assuming complete randomness of errors so that no particular pattern is more or less likely to be undetected than any other pattern. The EN 50159-2 is a related rail standard that assumes an open transmission system that could be subject to authentication error (i.e., open to masquerading attacks by untrusted nodes pretending to be trusted nodes).

- Section 2.3: The IEC 61508-2 addresses issues that include masquerade faults.

- Section 3.1: DeviceNet is an industrial controls network protocol that uses a 12-bit CRC.

- Section 3.2: The PROFIsafe is an industrial controls network protocol built upon PROFIBUS, which is intended to increase the protection of critical data. The PROFIsafe uses a 16-bit CRC for short messages (up to 12 bytes), and a 32-bit CRC for longer messages (up to 122 bytes). This is in addition to the standard 16-bit PROFIBUS frame check sequence (FCS) that provides HD = 4. Note that other nonauthoritative sources suggest there is also a 24-bit CRC available for use with PROFIsafe.

- Section 3.4: EsaLAN is a CAN-based system for safety-related applications. It appears to use the standard 15-bit CRC provided as part of the CAN protocol.

- Section 3.5: SafetyBUS p® is a safety critical bus built upon CAN. It uses a 16-bit CRC in data packets in addition to the CAN CRC.

- Section 3.7 Interbus Safety is a safety-related extension of the Interbus machine and industrial control protocol. It uses a 16-bit CRC per message.

It should be noted that for most of the network protocols discussed above, other countermeasures beyond CRCs are used to ensure end-to-end integrity. Additionally, the report discusses other protocols that are either not relevant to the topic of checksums and CRCs or are discussed elsewhere in this literature review.

A-5.7 Other Industrial Control Protocols.

The Highway Addressable Remote Transducer protocol uses an LRC (sometimes incorrectly called an XOR checksum) [A-48].

The Modicon bus (Modbus) ASCII protocol uses a two's complement addition checksum [A-49].

A-6 INTERNET AND IT APPLICATIONS.

A-6.1 Ethernet CRC.

Reference: IEEE Std 802.3-2000, "Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specification," 2000, pp. i–1515 [A-14].

The most commonly used 32-bit CRC is the IEEE 802.3 (Ethernet) CRC polynomial defined in this standard as:
x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1

A-6.2 Internet Checksum.

The Internet checksum is used to protect packet headers beyond the protection afforded by Ethernet CRCs. Because the checksum is in a fixed location and covers the packet length field, this guards against corruption of the length field, causing the apparent location of the FCS to be incorrect.

Request for comment (RFC) 1071 describes the computation of the Internet checksum used by IP, User Datagram Protocol, and Transmission Control Protocol (TCP), as well as computational speedup techniques [A-50]. The Internet checksum is a one's complement 16-bit addition checksum.

RFC 1146 expands the options for TCP packets to include an 8-bit Fletcher checksum (giving a 16-bit check sequence) and a 16-bit Fletcher checksum (giving a 32-bit check sequence) [A-51].

A-6.3 Stream Control Transmission Protocol Checksum Change.

Reference: Stone, J., Stewart, R., Otis, D., "Stream Control Transmission Protocol (SCTP) Checksum Change," RFC 3309, The Internet Society, September 2002 [A-52].

This reference recommends a change from the Adler-32 checksum to a 32-bit CRC (CRC-32c from [A-17]) because of poor performance of the Adler-32 checksum, based on the first author's thesis work.

A-6.4 Stone and Partridge Studies.

Reference: Stone, J., Greenwald, M., Partridge, C., and Hughes, J., "Performance of Checksums and CRC's Over Real Data," *IEEE/ACM Trans. Networking*, Vol. 6, No. 5, October 1998, pp. 529–543 [A-53].

This paper discusses real data and finds that data values are not evenly distributed. Real data often have many zero values, which biases the computation of checksums.

Reference: Stone, J. and Partridge, C., "When the CRC and TCP Checksum Disagree," *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stockholm, Sweden, Vol. 30, No. 4, October 2000, pp. 309–319 [A-54].

Stone and Partridge identify a host of errors on real desktop computers and networks that cause checksums and CRC effectiveness to be undermined [A-54]. Problems seen in operational systems included end-host software errors, router memory errors, and header compression errors. Undetected errors were found to be much more common than expected. It was concluded that any system using Ethernet and a TCP checksum that required high data integrity should also use an application checksum or CRC of some sort.

A-6.5 Ethernet Errors and 8B/10B Encoding.

Reference: James, L., Moore, A., Glick, M., and Bulpin, J., "Physical Layer Impact Upon Packet Errors," *Proceedings of the Passive and Active Measurement Workshop (PAM2006)*, April 2006 [A-55; 16 in main document].

This paper studied the frequency and severity of bit errors on optical gigabit Ethernet. While many avionics networks are copper instead of fiber, it seems reasonable that some of the same error-detection issues will be relevant.

Multibit errors are reasonably common and more common in general than expected. Of the data sampled, nearly 10% of the frames with errors had two different bytes in error and 2% had between 3–5 bytes in error. This is significantly more than expected using a random independent bit error assumption.

Error amplification is due to 8B/10B encoding, with which a single physical error bit can result in 1–4 data error bits when the 10-bit symbol is decoded into 8 data bits. This means that two physical bit errors in the wrong position may possibly create an error undetectable by the Ethernet CRC. The paper reports findings of 19 distinct undetectable error patterns in a 1518-byte codeword, which could be caused by just two bytes being in error.

A-7. OTHER LITERATURE.

A-7.1 Composite CRC and Checksum Performance.

Reference: Tran, E., Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol, M.S. Thesis, CMU ECE, May 1999 [A-56; 12 in main document].

Tran proposed using a checksum or CRC inside a data packet to mitigate problems found with CAN's CRC-based error detection. A study of one packet CRC polynomial chosen indicated that a combination of packet checksum and CAN CRC provided better error detection than a combination of a packet CRC with the CAN CRC. However, the interaction of combinations of different possible packet CRCs was not explored to determine if this is a general result or just a result for that particular CRC combination. The evaluation criterion used was Pud.

A-7.2 Composite CRC Performance.

Reference: Youssef, A., Arlat, J., Crouzet, Y., DeBonneval, A., Aubert, J., and Brot, P., "A High Integrity Error Checking Scheme for Communication Networks in Critical Control Systems," Rapport LAAS No 07306, LAAS-CNRS, Toulouse, March 2007 [A-57].

Youssef et al. proposed using alternating CRC polynomials to detect systematic hardware faults that would otherwise produce undetectable errors. The intent was to decrease the Pud for such errors. Evaluation was based on the fraction of undetected errors using randomly selected codewords for one of two polynomials evaluated against whether that codeword would be valid for the other polynomial.

A-7.3 Memory Errors.

Detecting errors in memory arrays requires an understanding of the likely patterns of errors for multibit upsets. In communication networks, error clusters are often presumed to be one-dimensional bursts (i.e., a burst error), but in two-dimensional memory arrays, a single large error source may have a two-dimensional footprint into the error array. For example, Constantinescu [A-58] provides an example that shows four two-dimensional adjacent bits subject to an intermittent value error due to a microscopic residue on the die at the shared corner of all four of those bits. The impact of such errors on CRC or checksum effectiveness would be affected by the interleaving of memory words (which data bits in the array map to which memory addresses).

Reasonable models for memory array errors do not appear to be published. While single-event upsets are often assumed to be the dominant error model, this may not be true for terrestrial applications. It may also be possible that error rates vary significantly, even among identical device types [A-59].

A-7.4 Problems Beyond CRC and Checksums.

The following references mention other factors that can compromise checksum or CRC performance, but are beyond the scope of the current study.

Funk reports various error detecting problems with HDLC, including a single bit error in a Flag character causing an undetectable error despite use of a HD = 4 CRC [A-60].

Fiorini et al. examined other potential issues with the HDLC protocol that can cause it to undermine the effectiveness of the CRC [A-61]. Of particular concern are bit errors in stuff bits, which can lead to long sequences of data bits logically slipping one position so that a pair of stuff bit errors cascades into a burst error of arbitrarily long length (this form of bit slip is in addition to the receiver bit slip reported in [A-59]). This paper also points out that adding a count value to a message does not necessarily help if the count itself suffers a bit error (later protocol designs protect against a count bit error via a header CRC). Tran [A-56] discusses similar vulnerabilities in the CAN protocol.

Bit encoding, stuffing, and other considerations can significantly affect the performance of achieved error-detection rates when used in combination with checksums or CRCs. Reference A-62 discusses this topic at length.

Conversely, techniques such as heartbeat messages, sequence numbers within data packets, repeated transmissions, and robust bit encoding can help improve network integrity beyond checksum and CRC coverage.

<u>A-8. REFERENCES</u>.

A-1. Maxino, T. and Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," *IEEE Transactions on Dependable and Secure Computing*, January–March 2009, pp. 59–72.

A-2. Tzou Chen, S.C. and Fang, G.S., "A Closed-Form Expression for the Probability of Checksum Violation," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 10, No. 7, July 1980, pp. 407–410.

A-3. Jiao, C. and Schwiebert, L., "Error Masking Probability of 1's Complement Checksums," *Proceedings of the 10th International Conference on Computer Communications and Networks*, Scottsdale, Arizona, October 2001, pp. 505–510.

A-4. Desaki, Y., Iwasaki, K., Miura, Y., and Yokota, D., "Double and Triple Error Detecting Capability of Internet Checksum and Estimation of Probability of Undetectable Error," *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1997, pp. 47–52.

A-5. Saxena, N.R. and McCluskey, E.J., "Analysis of Checksums, Extended-Precision Checksums, and Cyclic Redundancy Checks," *IEEE Transactions on Computers*, Vol. 39, No. 7, July 1990, pp. 969–975.

A-6. Prange, E., "Cyclic Error-Correcting Codes in Two Symbols," AFCRC-TN-57-103, ASTIA Document No. AD133749, Air Force Cambridge Research Center, 1957.

A-7. Peterson, W.W. and Brown, D.T., "Cyclic Codes for Error Detection," *Proceedings of the IRE*, Vol. 49, January 1961, pp. 228–234.

A-8. Feldmeier, D.C., "Fast Software Implementation of Error Detection Codes," *IEEE/ACM Transactions on Networking*, Vol. 3, No. 6, December 1995, pp. 640–651.

A-9. Ray, J. and Koopman, P., "Efficient High Hamming Distance CRCs for Embedded Networks," *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, Washington, DC, USA, *IEEE Computer Society*, 2006, pp. 3–12.

A-10. Braun, F. and Waldvogel, M., “Fast Incremental CRC Updates for IP Over ATM Networks,” *IEEE Workshop on High Performance Switching and Routing*, Dallas, Texas, 2001, pp. 48–52.

A-11. Wheal, M. and Miller, M., “The Probability of Undetected Error With Error Detection Codes,” *IREEECON, 19th International Electronics Convention and Exhibition*, Sydney, Australia, 1983, pp. 464–466.

A-12. Fujiwara, T., Kasami, T., Kitai, A., and Lin, S., “On the Undetected Error Probability for Shortened Hamming Codes,” *IEEE Transactions On Communications*, Vol. COM-33, No. 6, June 1985, pp. 570–574.

A-13. Fujiwara, T., Kasami, T., and Lin, S., “Error Detecting Capabilities of the Shortened Hamming Codes Adopted for Error Detection in IEEE Standard 802.3,” *IEEE Transactions On Communications*, Vol. 37, No. 9, September 1989, pp. 986–989.

A-14. IEEE Std 802.3-2000, “Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specification,” 2000, pp. i–1515.

A-15. Daniel, J., Costello, J., Hagenauer, J., Imai, H., and Wicker, S.B., “Applications of Error-Control Coding” (invited paper), *IEEE Transactions on Information Theory*, Vol. 38, No. 1, October 1998, pp. 2531–2560.

A-16. Sheinwald, D., Satran, J., Thaler, P., and Cavanna, V., “Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations,” IETF RFC 3385, September 2002.

A-17. Castagnoli, G., Brauer, S., and Herrmann, M., “Optimization of Cyclic Redundancy-Check Codes With 24 and 32 Parity Bits,” *IEEE Transactions on Communications*, Vol. 41, No. 6, 1993, pp. 883–892.

A-18. Castagnoli, G., Ganz, J., and Graber, P., “Optimum Cycle Redundancy-Check Codes With 16-Bit Redundancy,” *IEEE Transactions on Communications*, Vol. 38, No. 1, January 1990, pp. 111–114.

A-19. Chun, D. and Wolf, J., “Special Hardware for Computing the Probability of Undetected Error for Certain Binary CRC Codes and Test Results,” *IEEE Transactions on Communications*, Vol. 42, No. 10, October 1994, pp. 2769–2772.

A-20. Funk, G., “Determination of Best Shortened Linear Codes,” *IEEE Transactions on Communications*, Vol. 44, No. 1, January 1996, pp. 1–6.

A-21. Baicheva, T., Dodunekov, S., and Kazakov, P., “On the Cyclic Redundancy-Check Codes with 8-Bit Redundancy,” *Computer Communications*, Vol. 21, 1998, pp. 1030–1033.

A-22. Leung-Yan-Cheong, S., Barnes, E., and Friedman, D., "On Some Properties of the Undetected Error Probability of Linear Codes," *IEEE Transactions on Information Theory*, Vol. IT-25, No. 1, January 1979, pp. 110–112.

A-23. Baicheva, T., Dodunekov, S., and Kazakov, P., "Undetected Error Probability Performance of Cyclic Redundancy-Check Codes of 16-Bit Redundancy," *IEEE Proceedings on Communications*, Vol. 147, No. 5, October 2000, pp. 253–256.

A-24. Kazakov, P., "Fast Calculation of the Number of Minimum-Weight Words of CRC Codes," *IEEE Transactions on Information Theory*, Vol. 47, No. 3, March 2001, pp. 1190–1195.

A-25. Koopman, P., "32-Bit Cyclic Redundancy Codes for Internet Applications," *Proceedings of the 2002 International Conference on Dependable Systems and Networks,* Washington, DC, USA, *IEEE Computer Society*, 2002, pp. 459–472.

A-26. "Manual on Detailed Technical Specifications for the Aeronautical Telecommunication Network (ATN) Using ISO/OSI Standards and Protocols, Part 1–Air-Ground Applications," Doc 9880, International Civil Aviation Organization, Montreal, Canada. First Edition, 2010.

A-27. "Multi-Transmitter Data Bus Part 1 Technical Description," ARINC Specification 629P1-5, *Airlines Electronic Engineering Committee*, Annapolis, Maryland, March 31, 1999.

A-28. Haidinger, W., "Analysis of the CRC Polynomial Used in TTP/C," Research Report 11/2003, Real-Time Systems Group, Technical University of Vienna, Feb 19, 2003.

A-29. TTTech Computertechnik AG, Time Triggered Protocol TTP/C High-Level Specification Document, Protocol Version 1.1, specification ed. 1.4.3, November 2003.

A-30. "AFDX/ARINC 664 Tutorial," Document 700008_TUT-AFDX-EN_1000, TechSAT GmbH, Poing, 2008.

A-31. IEEE Std 1394-1995, "IEEE Standard for a High Performance Serial Bus," August 1996, ISBN 0-7381-1203-8.

A-32. *Digital Systems Validation*, Vol. 2, Chapter 18: Avionic Data Bus Integration Technology, DOT/FAA/CT-88/10, FAA Technical Center, Atlantic City, NJ, November 1993.

A-33. "Software Approval Guidelines Order 8110.49 Chg 1," Federal Aviation Administration, 9/28/2011.

A-34. Paulitsch, M., Morris, J., Hall, B., et al., "Coverage and Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005, pp. 346–355.

A-35. International Standard ISO 26262-5:2011(E), "Road vehicles – Functional Safety – Part 5: Product Development at the Hardware Level," First Edition, International Standards Organization, Geneva, Switzerland, November 15, 2011.

A-36. Koopman, P. and Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," *Proceedings of International Conference on Dependable Systems and Networks*, Florence, Italy, June 2004, pp. 145–154.

A-37. "FlexRay Communications System Protocol Specification Version 3.0.1," *FlexRay Consortium*, 2010.

A-38. "Specification of SW-C End-to-End Communication Protection Library," V 1.1.0, R4.0 Rev 2, AUTOSAR Standard, 12/17/2009.

A-39. Mehrnoush, R., Muller-Rathgeber, B., and Steinbach, E, "Error Detection Capabilities of Automotive Network Technologies and Ethernet – A Comparative Study," *Proceedings of the IEEE Intelligent Vehicles Symposium*, Istanbul, Turkey, June 2007, pp. 674–679.

A-40. Robert Bosch GmbH, "CAN Specification Version 2.0," Sept. 1991.

A-41. "OpenSAFETY: The First Open and Bus-Independent Safety Standard for All Industrial Ethernet Solutions," available at http://open-safety.org/ (accessed January 25, 2012).

A-42. "Software Package for the Development of Safety Nodes (SN) and Safety Configuration Manager (SCM), http://www.ixxat.com/ethernet_powerlink_safety_stack_en.html (accessed on January 25, 2012).

A-43. "Profibus–DP/PA: Profisafe–Profile for Failsafe Technology, v 1.0, Document No. 740257," March 1999, p. 33.

A-44. "Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems," SoHar Incorporated, US Nuclear Regulatory Commission, Washington DC, June 1996.

A-45. "Data Communications," Lawrence Livermore National Laboratory, U.S. Nuclear Regulatory Commission, Washington, DC, August 1993.

A-46. Koopman, P. and Chakravarty, T., "Analysis of the Train Communication Network Protocol Error Detection Capabilities," Carnegie Mellon University Technical Report, February 25, 2001.

A-47. Alanen, J., Hietikko, M., and Malm, T., "Safety of Digital Communications in Machines," VTT Research Notes 2265, VTT Industrial Systems, 2004.

A-48. "The HART™ Book, The HART Message Structure, What Is HART?" December 2005.

A-49. "Modbus Protocol Reference Guide," Modicon, Inc., pI-MBUS-300 Rev. J., June 1996.

A-50. Braden, R., Borman, D., and Partridge, C., "Computing the Internet Checksum," IETF RFC 1071, September 1988.

A-51. Zweig, J. and Partridge, C., "TCP Alternate Checksum Options," IETF RFC 1146, March 1990.

A-52. Stone, J., Stewart, R., and Otis, D.,, "Stream Control Transmission Protocol (SCTP) Checksum Change," RFC 3309, The Internet Society, September 2002.

A-53. Stone, J., Greenwald, M., Partridge, C., and Hughes, J., "Performance of Checksums and CRCs over Real Data," *IEEE/ACM Transactions on Networking*, Vol. 6, No. 5, October 1998, pp. 529–543.

A-54. Stone, J. and Partridge, C., "When the CRC and TCP Checksum Disagree," *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stockholm, Sweden, Vol. 30, No. 4, October 2000, pp. 309–319.

A-55. James, L., Moore, A., Glick, M., and Bulpin, J., "Physical Layer Impact Upon Packet Errors," *Proceedings of the Passive and Active Measurement Workshop*, April 2006.

A-56. Tran, E., "Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol," M.S. Thesis, CMU ECE, May 1999.

A-57. Youssef, A., Arlat, J., Crouzet, Y., DeBonneval, A., Aubert, J., and Brot, P., "A High Integrity Error Checking Scheme for Communication Networks in Critical Control Systems," Rapport LAAS No 07306, March 2007.

A-58. Constantinescu, C., "Intermittent Faults and Effects on Reliability of Integrated Circuits," *Reliability and Maintainability Symposium*, Las Vegas, Nevada, January 2008, pp. 370–374.

A-59. Schroeder, B., Pinheiro, E., Weber, W-D., "DRAM Errors in the Wild: A Large-Scale Field Study," *SIGMETRICS09*, 2009, pp. 193–204.

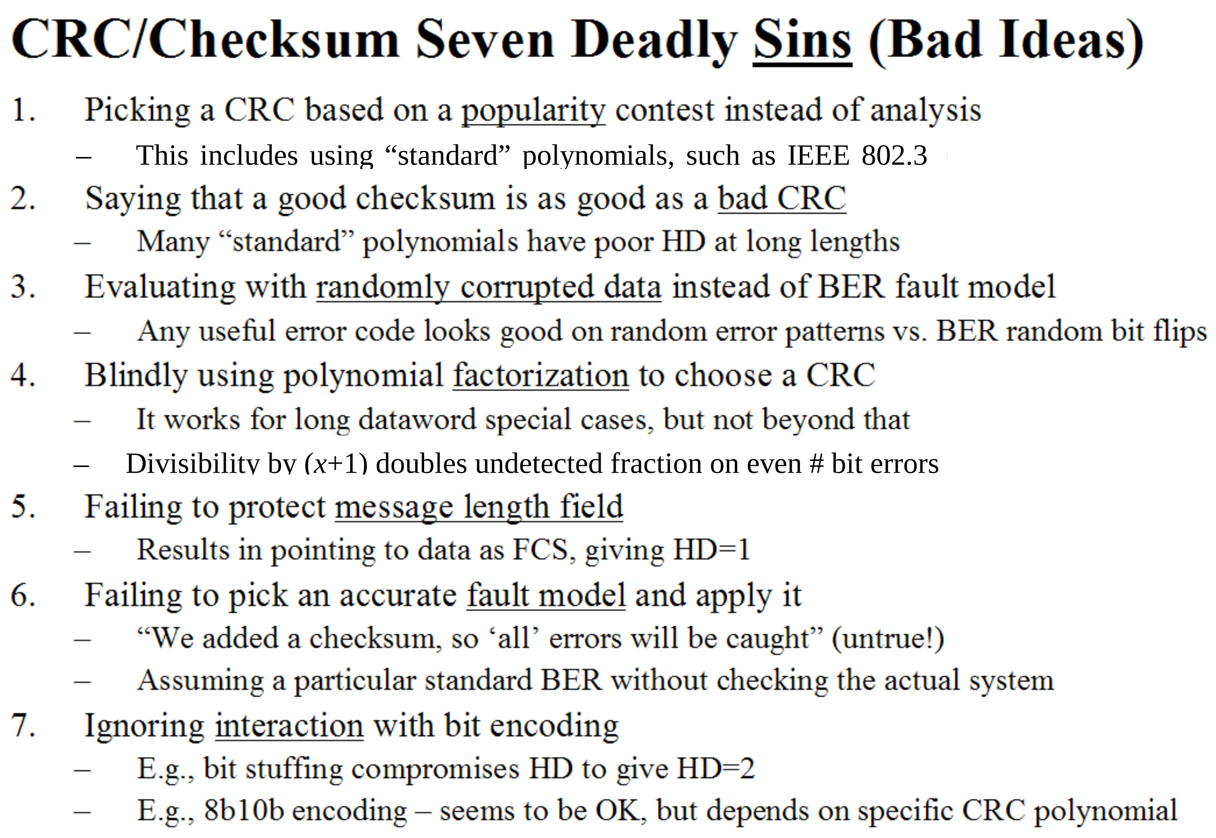
A-60. Funk, G., "Message Error Detecting Properties of HDLC Protocols," *IEEE Transactions on Communciations*, Vol. COM-30, No. 1, January 1982, pp. 252–257.

A-61. Fiorini, D., Chiani, M., Tralli, V., and Salati, C., "Can We Trust in HDLC?" *ACM SIGCOMM Computer Communication Review*, Vol. 24, No. 5, October 1994, pp. 61–80.

A-62. "Impact of Bit Coding on Error Rates," ISO/TC22/SC3/WG 1, *Task Force: Parameters to be Standardized French Experts Contribution*, Detroit, Michigan, February 27 and 28, 1989.

APPENDIX B—SEVEN DEADLY SINS FOR CHECKSUMS AND CYCLIC REDUNDANCY CODES

The following is a list of poor practices that are commonly committed, but should be avoided, when designing a system with cyclic redundancy codes (CRCs) and checksums.



CRC/Checksum Seven Deadly Sins (Bad Ideas)

1. Picking a CRC based on a popularity contest instead of analysis
   – This includes using “standard” polynomials, such as IEEE 802.3
2. Saying that a good checksum is as good as a bad CRC
   – Many “standard” polynomials have poor HD at long lengths
3. Evaluating with randomly corrupted data instead of BER fault model
   – Any useful error code looks good on random error patterns vs. BER random bit flips
4. Blindly using polynomial factorization to choose a CRC
   – It works for long dataword special cases, but not beyond that
   – Divisibility by $(x+1)$ doubles undetected fraction on even # bit errors
5. Failing to protect message length field
   – Results in pointing to data as FCS, giving HD=1
6. Failing to pick an accurate fault model and apply it
   – “We added a checksum, so ‘all’ errors will be caught” (untrue!)
   – Assuming a particular standard BER without checking the actual system
7. Ignoring interaction with bit encoding
   – E.g., bit stuffing compromises HD to give HD=2
   – E.g., 8b10b encoding – seems to be OK, but depends on specific CRC polynomial

APPENDIX C—CYCLIC REDUNDANCY CODE AND CHECKSUM TUTORIAL SLIDES

The following pages are the handouts from the cyclic redundancy code and checksum tutorial webinar conducted as part of this research.

Tutorial: Checksum and CRC Data Integrity Techniques for Aviation

May 9, 2012

Philip Koopman
Carnegie Mellon University
koopman@cmu.edu

Co-PIs:
Kevin Driscoll
Brendan Hall
Honeywell Laboratories

The views and opinions expressed in this presentation are those of the author, and are not necessarily those of the Federal Aviation Administration, who sponsored this work under contract DTFACT-11-C-00005. This presentation does not contain any technical conclusions funded by this work.

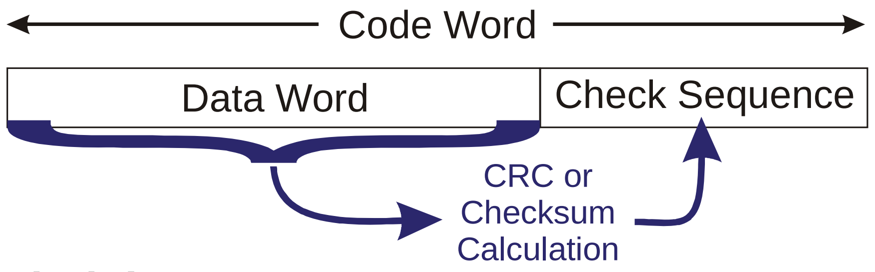
Agenda

- Introduction
  - Motivation – why isn't this a solved problem?
  - Parity computations as an example
  - Error code construction and evaluation (without scary math)
  - Example using parity codes
- Checksums
  - What's a checksum?
  - Commonly used checksums and their performance
- Cyclic Redundancy Codes (CRCs)
  - What's a CRC?
  - Commonly used CRC approaches and their performance
- Don't blindly trust what you hear on this topic
  - A good CRC is almost always **much** better than a good Checksum
  - Many published (and popular) approaches are suboptimal or just plain wrong
  - There are some topics to be careful of because we don't know the answers

- Q&A

Checksums and CRCs Protect Data Integrity

- Compute check sequence when data is transmitted or stored
  - **Data Word**: the data you want to protect (can be any size; often Mbytes)
  - **Check Sequence**: the result of the CRC or checksum calculation
  - **Code Word** = Data Word with Check Sequence Appended



- To check data integrity:
  - Retrieve or receive Code Word
  - Compute CRC or checksum on the received Data Word
  - If computed value equals Check Sequence then no data corruption found
    - (There might be data corruption! But if there is, you didn't detect it.)

Potential CRC/Checksum Usage Scenarios

- Network packet integrity check

- Image integrity check for software update
- Boot-up integrity check of program image
  - e.g., flash memory data integrity check
- FPGA configuration integrity check

- Configuration integrity check
  - e.g., operating parameters in EEPROM
- RAM value integrity check

Why Is This A Big Deal?

- Checksums are pretty much as good as CRCs, right?
  - In a word – **NO!**
  - Typical studies of checksums compare them to horrible CRCs
  - Would you prefer to detect all 1 & 2-bit errors (checksum) or all possible 1, 2, 3, 4, 5-bit errors (CRC) for about the same cost?
- CRCs have been around since 1957 – aren't they a done deal?
  - In a word – **NO!**
  - There wasn't enough compute power to find optimal CRCs until recently… so early results are often *not* very good
  - There is a lot of incorrect writing on this topic … that at best assumes the early results were good
  - Many widespread uses of CRCs are mediocre, poor, or broken
- Our goal today is to show you where the state of the art really is
  - And to tune up your sanity check detector on this topic
  - Often you can get ***many orders of magnitude better error detection*** simply by using a good CRC at about the same cost

Error Coding For Poets **(who know a little discrete math)**

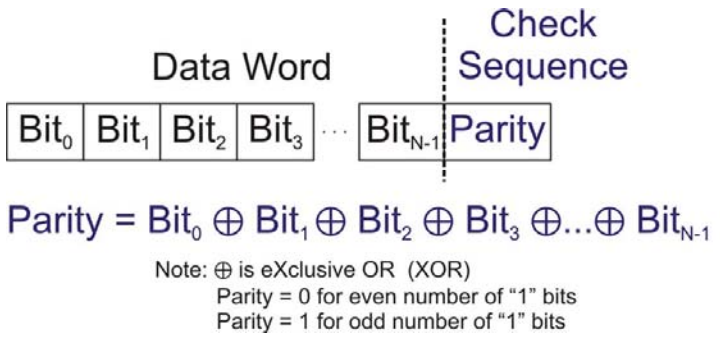
- The general idea of an error code is to mix all the bits in the data word to produce a condensed version (the check sequence)
  - Ideally, every bit in the data word affects many check sequence bits
  - Ideally, bit errors in the code word have high probability of being detected
  - Ideally, more probable errors with only a few bits inverted detected 100% of the time
  - At a hand-wave, similar to desired properties of a pseudo-random number generator
    - The Data Word is the seed value, and the Check Sequence is the pseudo-random number

Data Word	Check Sequence

- The ability to do this will depend upon:
  - **The size of the data word**
    - Larger data words are bigger targets for bit errors, and are harder to protect
  - **The size of the check sequence**
    - More check sequence bits makes it harder to get unlucky with multiple bit errors
  - **The mathematical properties of the "mixing" function**
    - Thorough mixing of data bits lets the check sequence detect simple error patterns
  - **The type of errors you expect to get** (patterns, error probability)

Example: Parity As An Error Detection Code

- Example error detection function: **<u>Parity</u>**
  - XOR all the bits of the data word together to form 1 bit of parity

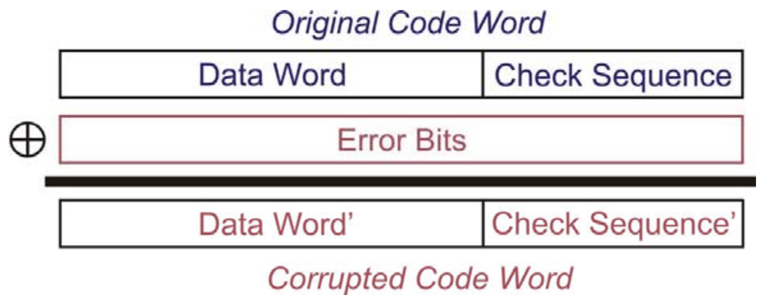

Data Word
Check Sequence
Bit$_0$ Bit$_1$ Bit$_2$ Bit$_3$ … Bit$_{N-1}$ Parity
Parity = Bit$_0$ ⊕ Bit$_1$ ⊕ Bit$_2$ ⊕ Bit$_3$ ⊕…⊕ Bit$_{N-1}$
Note: ⊕ is eXclusive OR (XOR)
Parity = 0 for even number of "1" bits
Parity = 1 for odd number of "1" bits

XOR Facts:
$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$
(Think of it as Boolean addition and subtraction.)

- How good is this at error detection?
  - Only costs one bit of extra data; all bits included in mixing
  - Detects all odd number of bit errors (1, 3, 5, 7, … bits in error)
  - Detects NO errors that flip an even number of bits (2, 4, 6, … bits in error)
  - Performance: detects up to 1 bit errors; misses all 2-bit errors
  - Not so great – can we do better?

Basic Model For Data Corruption

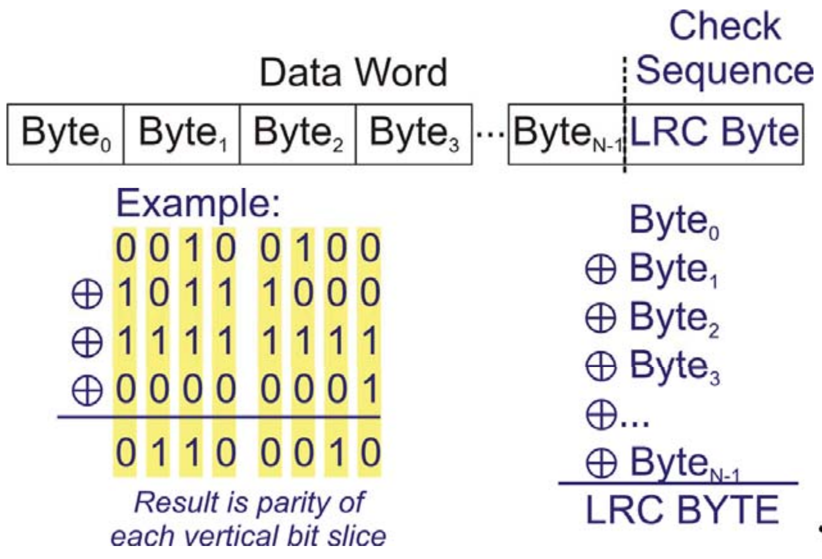
- Data corruption is "bit flips" ("bisymmetric inversions")
  - Each bit has some probability of being inverted
  - **"Weight"** of error word is number of bits flipped (number of "1" bits in error)



- Error detection works if the corrupted Code Word is invalid
  - In other words, if corrupted Check Sequence doesn't match the Check Sequence that would be computed based on the Data Word
  - If corrupted Check Sequence just happens to match the Check Sequence computed for corrupted data, you have an **<u>undetected error</u>**
  - All things being equal (which they are **<u>not!!!</u>**) probability of undetected error is 1 chance in 2^k for a k-bit check sequence

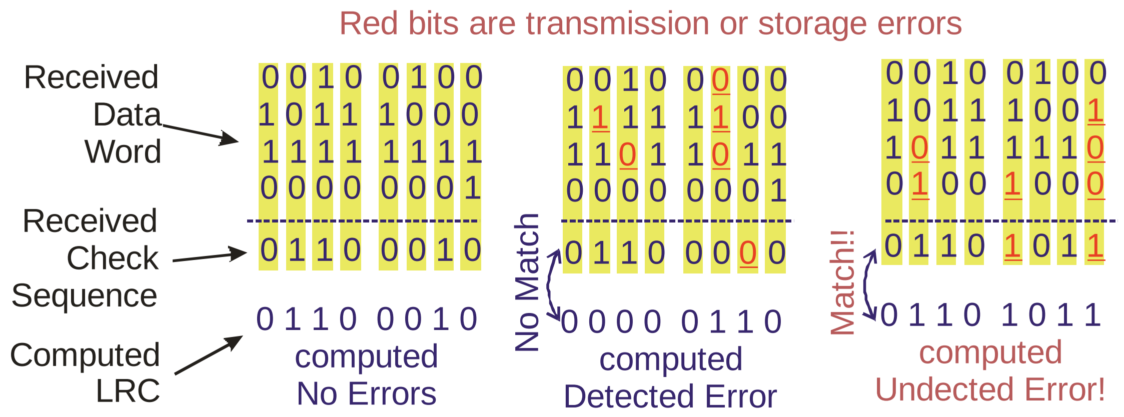
Example: Longitudinal Redundancy Check (LRC)

- LRC is a byte-by-byte parity computation
  - XOR all the bytes of the data word together, creating a one-byte result
  - (This is sometimes called an “XOR checksum”
    but it isn’t really integer addition, so it’s not quite a “sum”)



How Good Is An LRC?

- Parity is computed for each bit position (vertical stripes)
  - Note that the **received copy of check sequence** can be corrupted too!



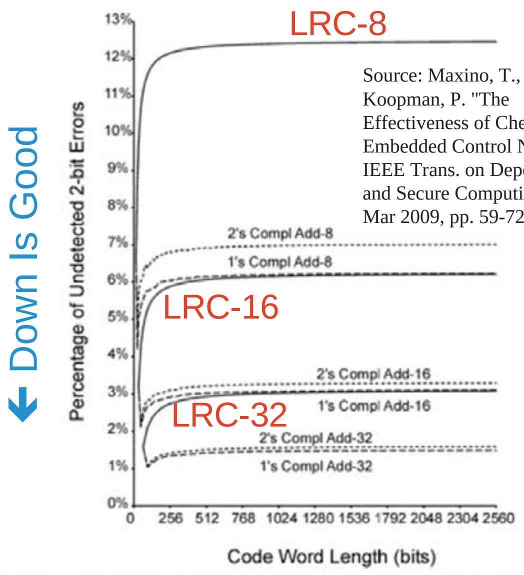
- Detects all odd numbers of bit errors in a vertical slice
  - Fails to detect even number of bit errors in a vertical slice
  - Detects all 1-bit errors; Detects all errors within a single byte
  - Detects many 2-bit errors, <u>**but not *all* 2-bit errors**</u>
    - Any 2-bit error in same vertical slice is undetected

Error Code Effectiveness Measures

- Metrics that matter depend upon application, but usual suspects are:
  - **Maximum weight** of error word that is 100% detected
    - Hamming Distance (HD) is lowest weight of any undetectable error
    - For example, HD=4 means all 1, 2, 3 bit errors detected
  - **Fraction of errors** undetected for a given number of bit flips
    - Hamming Weight (HW): how many of all possible m-bit flips are undetected?
      - E.g. HW(5)=157,481 undetected out of all possible 5-bit flip Code Word combinations
  - **Fraction of errors** undetected at a given random probability of bit flips
    - Assumes a **Bit Error Ratio (BER)**, for example 1 bit out of 100,000 flipped
    - Small numbers of bit flips are most probable for typical BER values
  - **Special patterns** 100% detected, such as adjacent bits
    - Burst error detection – e.g., all possible bit errors within an 8 bit span
  - Performance usually depends upon data word size and code word size
- Example for LRC8 (8 bit chunk size LRC)
  - HD=2 (all 1 bit errors detected, not all 2 bit errors)
  - Detects all 8 bit bursts (only 1 bit per vertical slice)
  - Other effectiveness metrics coming up…

LRC-8 Fraction of Undetected Errors

- Shows Probability of Undetected 2-bit Errors for:
  - LRC
  - Addition checksum
  - 1's complement addition checksum
- 8-bit addition checksum is almost as good as 16-bit-LRC!
  - So we can do better for sure

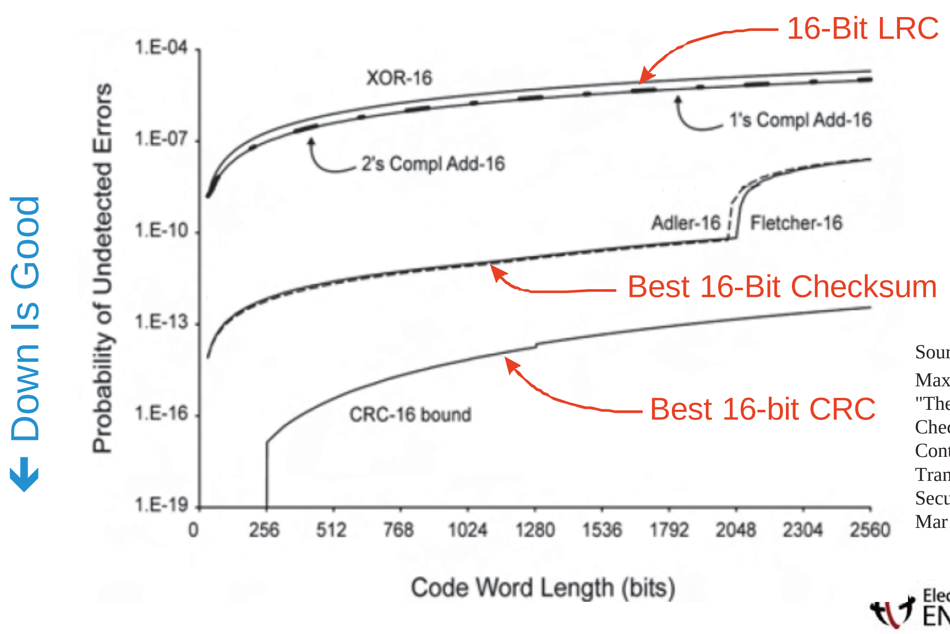


Source: Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

Fig. 1. Percentage of undetected 2-bit errors over the total number of 2-bit errors for 8-, 16-, and 32-bit XOR, two's complement addition, and one's complement addition checksums. Two's complement addition and one's complement addition data values are the mean of 100 trials using random data.

Can We Do Even Better?   YES!

- Can often get HD=6 (detect all 1, 2, 3, 4, 5-bit errors) with a CRC
- For this graph, assume Bit Error Rate (BER) = 10^{-5} flip probability per bit



Source:
Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

Checkpoint – What’s Coming Next

- You now have basic vocabulary and background

- Let’s talk about better ways to detect errors
  - Checksums
  - Cyclic Redundancy Codes (CRCs)
  - Evaluation strategies
  - Pitfalls

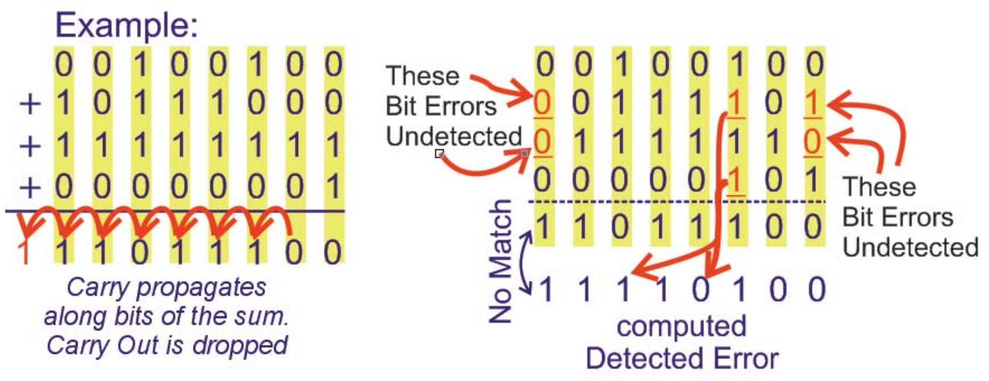
Checksums

- A checksum "adds" together "chunks" of data
  - The "add" operation may not be normal integer addition
  - The chunk size is typically 8, 16, or 32 bits

- We'll discuss:
  - Integer addition "checksum"
  - One's complement "checksum"
  - Fletcher Checksum
  - Adler Checksum
  - ATN Checksum (AN/466)

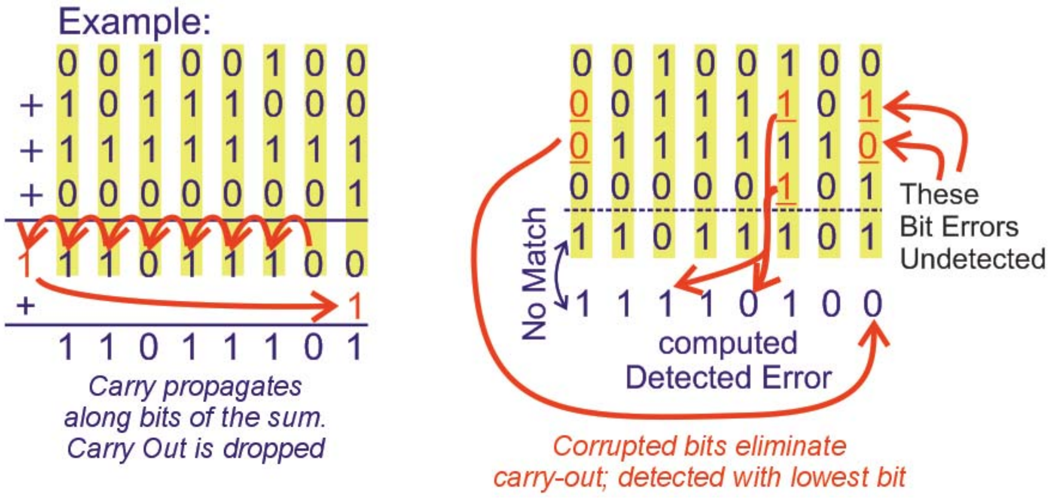
Integer Addition Checksum

- Same as LRC, except use integer "+" instead of XOR
  - The carries from addition promote bit mixing between adjacent columns
    - Can detect errors that make two bits go 0 ➔ 1 or 1 ➔ 0 (except top-most bits)
    - Cannot detect compensating errors (one bit goes 0 ➔1 and another 1 ➔ 0)
  - Carry out of the top bit of the sum is discarded
    - No pairs of bit errors are detected in top bit position



One's Complement Addition Checksum

- Same as integer checksum, but add Carry-Out bits back
  - Plugs error detection hole of two top bits flipping with the same polarity
  - But, doesn't solve problem of compensating errors
  - Hamming Distance 2 (HD=2); some two-bit errors are undetected



Fletcher Checksum

- Use two running one's complement checksums
  - For fair comparison, each running sum is half width
  - E.g., 16-bit Fletcher Checksum is two 8-bit running sums

  - Initialize: A = 0; B = 0;
  - For each byte in data word: A = A + Byte_i; B = B + A;
    - One's complement addition!
  - Result is A concatenated with B (16-bit result)

- Significant improvement comes from the running sum B
  - B = $\text{Byte}_{N-1} + 2*\text{Byte}_{N-2} + 3*\text{Byte}_{N-3} + \ldots$
  - Makes checksum order-dependent (switched byte order detected)
  - **Gives HD=3** until the B value rolls over
    - For example, $256*\text{Byte}_{N-256}$ does not affect B

Adler Checksum

- Intended to be an improvement on Fletcher Checksum
  - One's complement addition is the same as modulo 255 addition
  - Adler checksum uses a prime integer as a modulus
    - 251 instead of 255 for Adler 16 (two 8-bit sums)
    - 65521 instead of 65535 for Adler 32 (two 16-bit sums)

- In practice, it is not worth it
  - For most sizes and data lengths Adler is worse than Fletcher
  - In the best case it is only very slightly better
    - But computation is more expensive because of the modular sum

ATN-32 Checksum [AN/466]

- Aviation-specific riff on Fletcher Checksum
  - Four running 1-byte sums (one's complement addition)
  - Potentially gives good mixing for 8-bit data chunks

- Algorithm:
  - Initialize C0, C1, C2 and C3 to zero
  - For each Data Word byte:

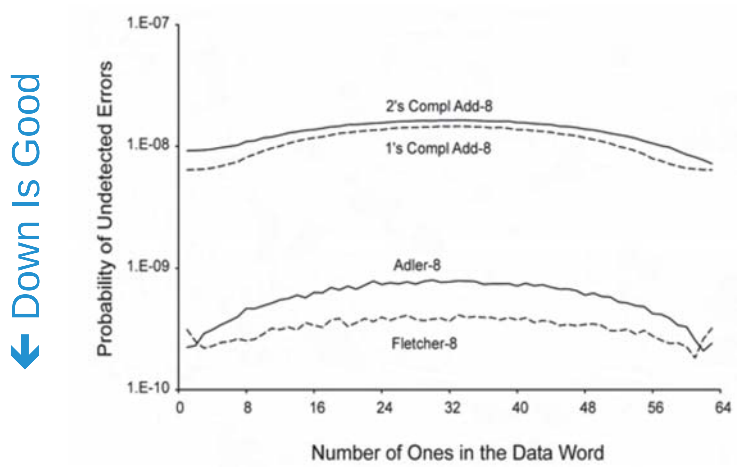
    C0 += Byte_i; C1 += C0; C2 += C1; C3 += C2;

    (one's complement addition, as with Fletcher checksum)
  - 32-bit check sequence is a particular formula of C0..C3

- No apparent published analysis of error detection results
    - Standard says it provides good protection, but no quantitative assessment
    - We'll take a look at this and other relevant error codes in our study

Checksum Performance Is Data Dependent

- The data values affect checksum performance
  - Worst-case performance is equal number of zeros and ones
  - Below is 64-bit data word and BER of 10^{-5}



Source:
Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

- This means need to take into account data values when assessing performance

Cyclic Redundancy Codes (CRCs)

- **CRCs Use Division Instead Of Addition**
- Intuitive description:
  - Addition does OK but not great mixing of Data Word bits
  - What about using the remainder after division instead?
- Integer analogy: remainder after integer division
  - 2,515,691,591 mod 251 = 166 🡐 8-bit check sequence
    - Any simple change to the input number (Data Word) changes remainder
  - But, need to pick a clever divisor
    - E.g., 2,515,691,591 mod 100 = 91 🡐 unaffected by most digits
    - Probably want something like prime number 251, but may be more complex than that to avoid "wasting" result values of 252, 253, 254, 255
  - ISBNs use this technique for the last digit, with divisor of 11
    - An "X" at the end of an ISBN means the remainder was 10 instead of 0..9
  - Also, want something that is efficient to do in SW & HW
    - Original CRCs were all in hardware to maximize speed and minimize hardware cost

Mathematical Basis of CRCs

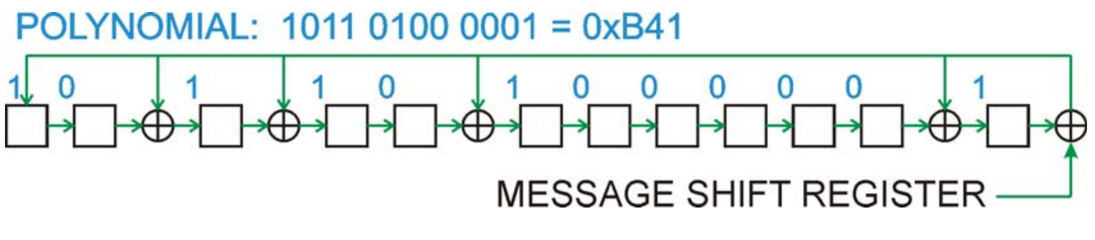
- Use polynomial division (remember that from high school?) over Galois Field(2) (this is a mathematician thing)
  - At a hand-waving level this is division using Boolean Algebra
    - "Add" and "Subtract" used by division algorithm both use XOR

```
11010011101100 000 <--- Data Word left shifted by 3 bits
1011               <--- 4-bit divisor is 1011  x³ + x + 1
01100011101100 000 <--- result of first conditional subtraction
 1011              <--- divisor
00111011101100 000 <--- result of second conditional subtraction
  1011             <--- continue shift-and-subtract ...
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------   Remainder is the Check Sequence
00000000000000 100 <--- Remainder (3 bits)
```

[Wikipedia]

Hardware View of CRC

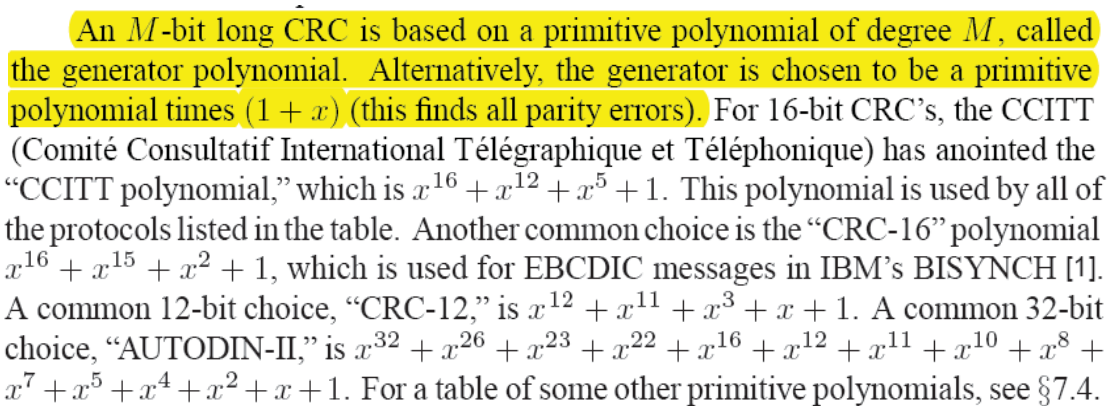
- CRC also has a clever hardware implementation:
  - The feedback "polynomial" is the divisor; shift register holds remainder



Example Feedback Polynomial:
0xB41 = x12+x10+x9+x7+x+1 (the "+1" is implicit in the hex value)
= (x+1)(x3 +x2 +1) (x8 +x4 +x3 +x2 +1)

- The tricky part is in picking the right Feedback Polynomial (divisor)
  - The best ones are not necessarily "prime" (irreducible) nor "primitive"
  - A lot of what is published on this topic has problems

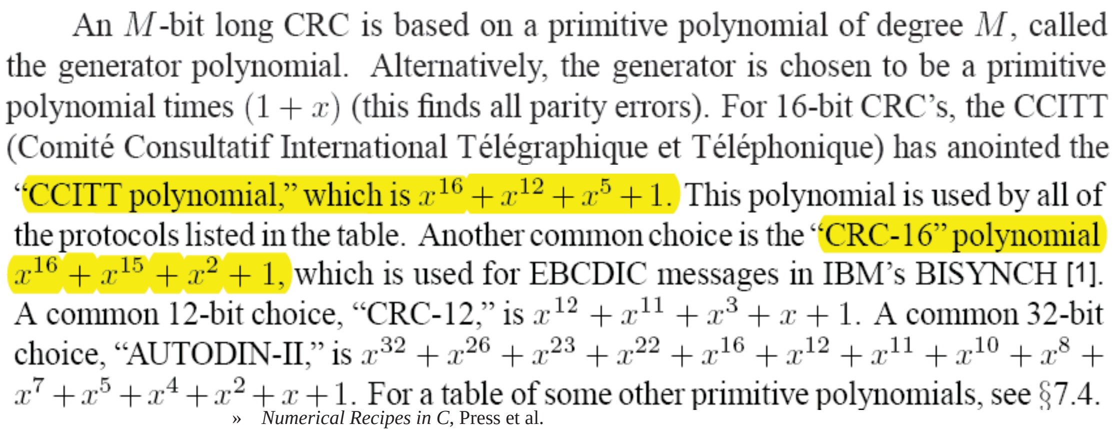
A Typical Legacy CRC Selection Method


An M-bit long CRC is based on a primitive polynomial of degree M, called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times $(1 + x)$ (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is $x^{16} + x^{12} + x^5 + 1$. This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial $x^{16} + x^{15} + x^2 + 1$, which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is $x^{12} + x^{11} + x^3 + x + 1$. A common 32-bit choice, "AUTODIN-II," is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. For a table of some other primitive polynomials, see §7.4.

» ***Numerical Recipes in C*, Press et al. 1992**

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by (x+1)
  - Divisibility by (x+1) doubles undetected error rate for even # of bit errors

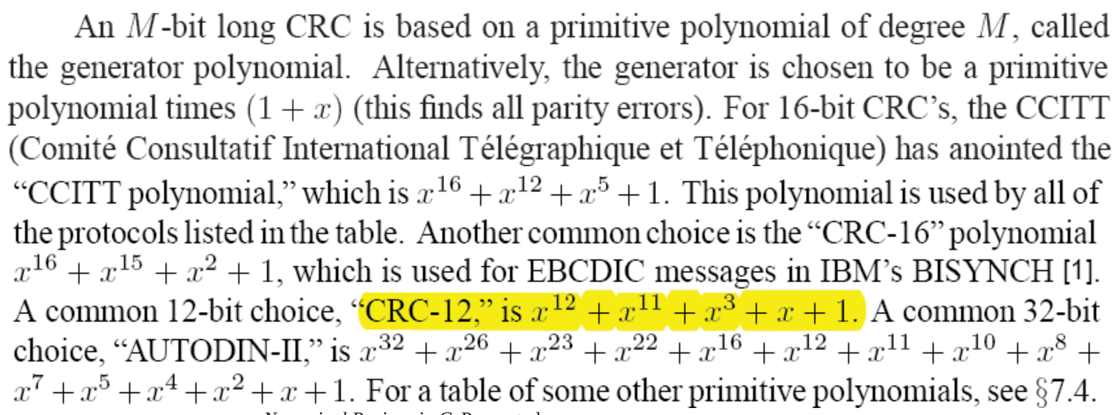
A Typical Polynomial Selection Method


An M-bit long CRC is based on a primitive polynomial of degree M, called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times $(1 + x)$ (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is $x^{16} + x^{12} + x^5 + 1$. This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial $x^{16} + x^{15} + x^2 + 1$, which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is $x^{12} + x^{11} + x^3 + x + 1$. A common 32-bit choice, "AUTODIN-II," is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. For a table of some other primitive polynomials, see §7.4.

» *Numerical Recipes in C*, Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by (x+1)
  - Divisibility by (x+1) doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?

A Typical Polynomial Selection Method

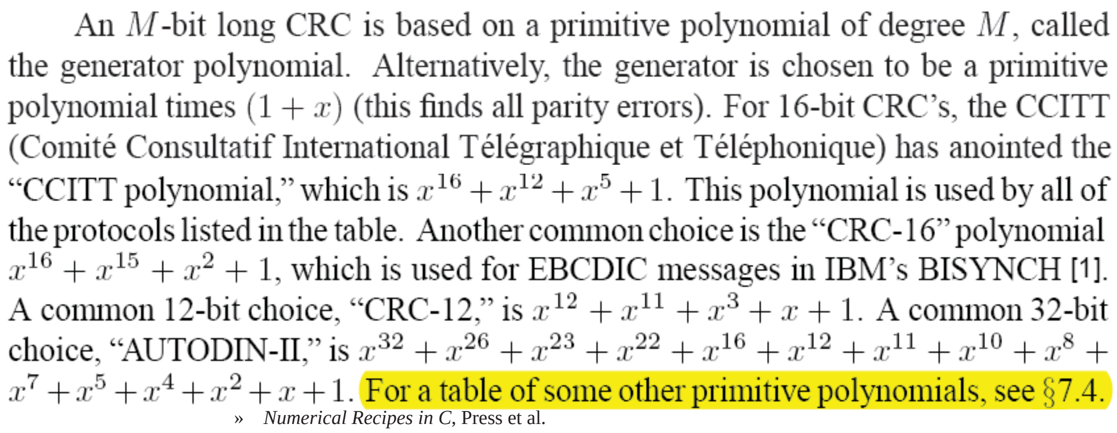


An M-bit long CRC is based on a primitive polynomial of degree M, called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times $(1 + x)$ (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is $x^{16} + x^{12} + x^5 + 1$. This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial $x^{16} + x^{15} + x^2 + 1$, which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is $x^{12} + x^{11} + x^3 + x + 1$. A common 32-bit choice, "AUTODIN-II," is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. For a table of some other primitive polynomials, see §7.4.

» *Numerical Recipes in C,* Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by (x+1)
  - Divisibility by (x+1) doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?
  - This CRC-12 polynomial is incorrect (there is a missing +x^2)

A Typical Polynomial Selection Method



An M-bit long CRC is based on a primitive polynomial of degree M, called the generator polynomial. Alternatively, the generator is chosen to be a primitive polynomial times $(1 + x)$ (this finds all parity errors). For 16-bit CRC's, the CCITT (Comité Consultatif International Télégraphique et Téléphonique) has anointed the "CCITT polynomial," which is $x^{16} + x^{12} + x^5 + 1$. This polynomial is used by all of the protocols listed in the table. Another common choice is the "CRC-16" polynomial $x^{16} + x^{15} + x^2 + 1$, which is used for EBCDIC messages in IBM's BISYNCH [1]. A common 12-bit choice, "CRC-12," is $x^{12} + x^{11} + x^3 + x + 1$. A common 32-bit choice, "AUTODIN-II," is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. For a table of some other primitive polynomials, see §7.4.

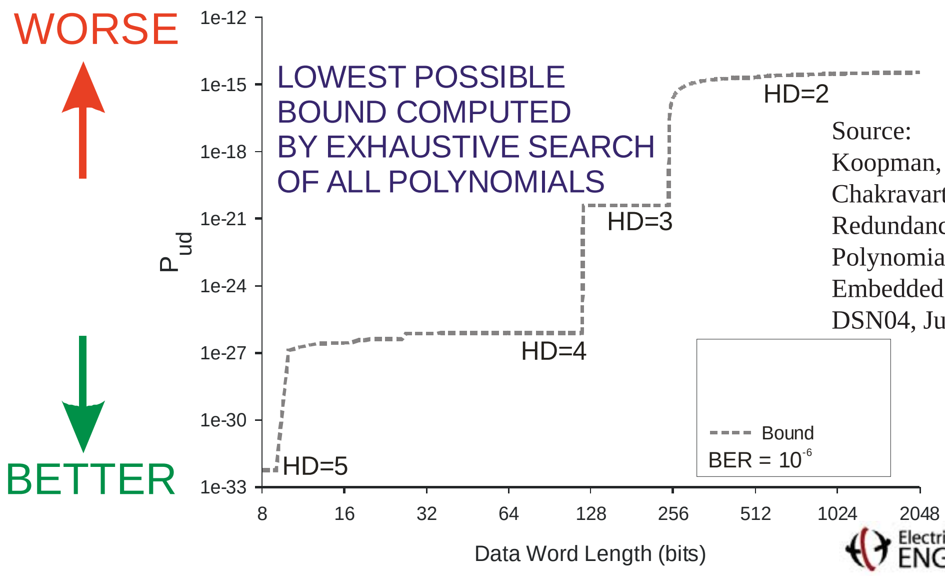
» *Numerical Recipes in C,* Press et al.

- But, there are some problems:
  - Many good polynomials are not primitive nor divisible by (x+1)
  - Divisibility by (x+1) doubles undetected error rate for even # of bit errors
  - How do you know which competing polynomial to pick?
  - This CRC-12 polynomial is incorrect (there is a missing +x^2)
  - You can't pick at random from a list!

(BTW, 3rd edition has updated this material and gets it right)

Example – 8 Bit Polynomial Choices

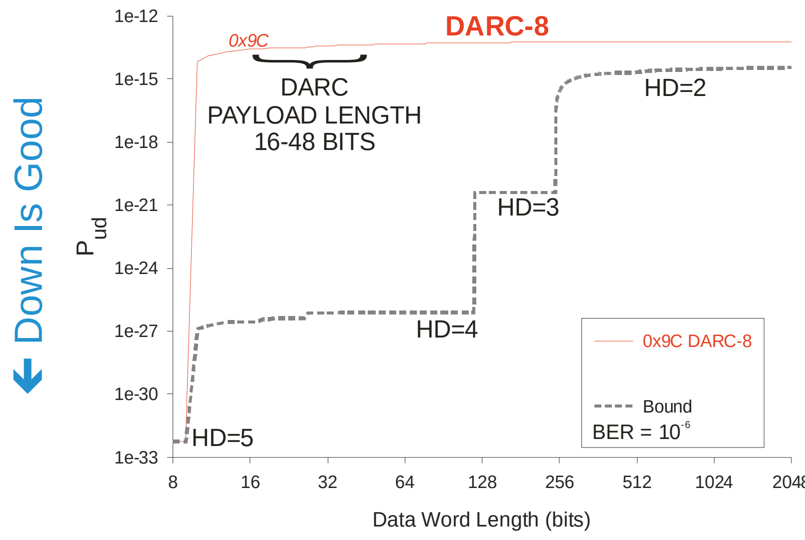
- P_{ud} (undetected error rate) is one way to evaluate CRC effectiveness
  - Uses Hamming weights of polynomials
  - Uses assumed random independent Bit Error Rate (BER)



Source: Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,“ DSN04, June 2004

What Happens When You Get It Wrong?

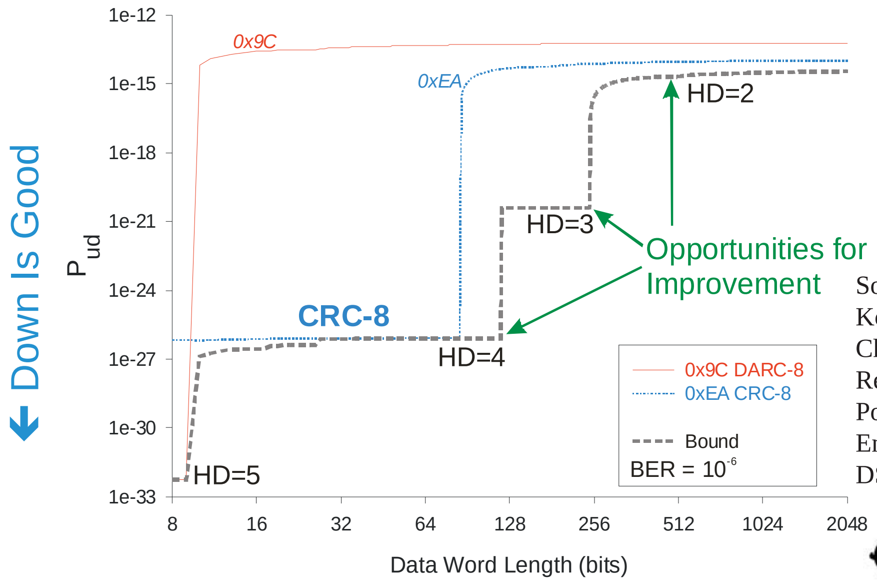
- DARC (Data Radio Channel), ETSI, October 2002
  - DARC-8 polynomial is optimal for 8-bit payloads
  - BUT, DARC uses 16-48 bit payloads, and misses some 2-bit errors
  - Could have detected all 2-bit and 3-bit errors with same size CRC!



Source: Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,“ DSN04, June 2004

CRC-8 Is Better

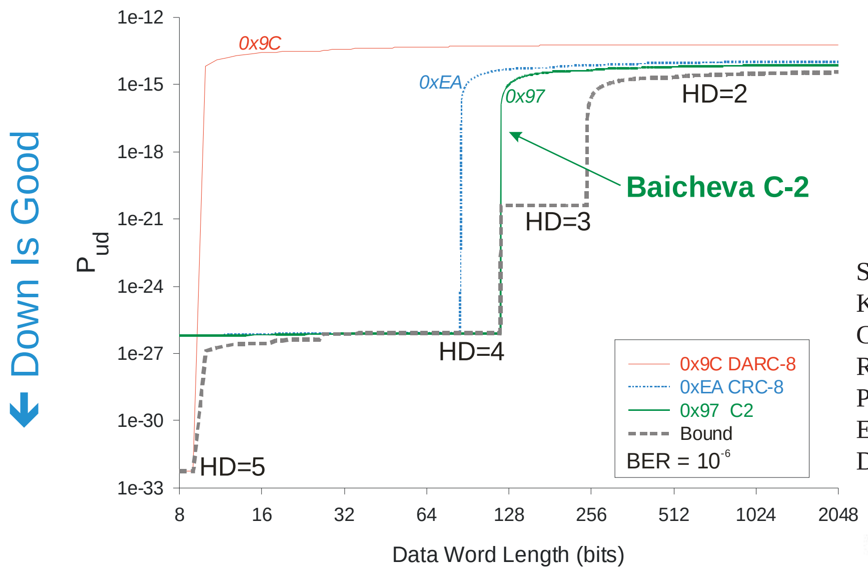
- CRC-8 (0xEA) is in very common use
  - Good for messages up to size 85
  - But, room for improvement at longer lengths. Can we do better?



Source:
Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,“ DSN04, June 2004

Baicheva's Polynomial C2

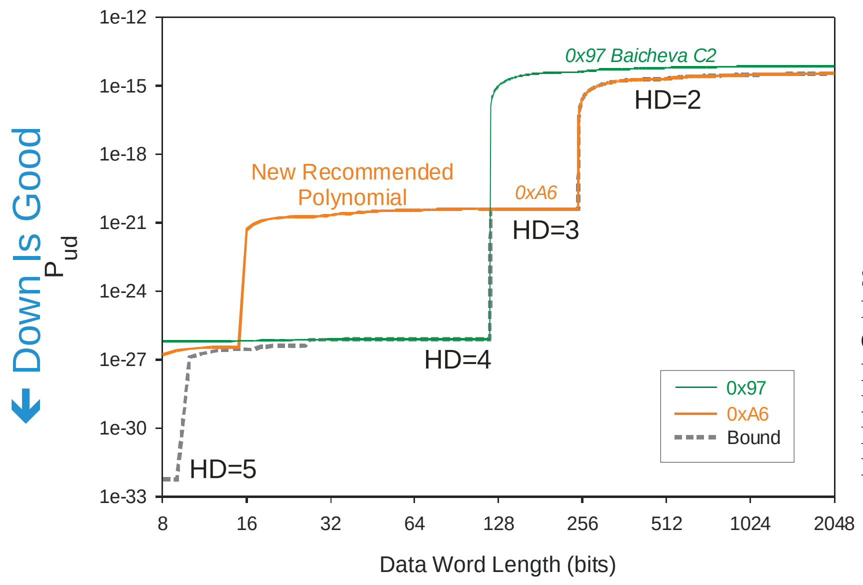
- [Baicheva98] proposed polynomial C2, 0x97
  - Recommended as good polynomial to length 119
  - Dominates 0xEA (better P_{ud} at every length)



Source:
Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,“ DSN04, June 2004

But What If You Want the HD=3 Region?

- No previously published polynomials proposed for HD=3 region
  - We found that 0xA6 has good performance
  - Better than C2 and near optimal at all lengths of 120 and above

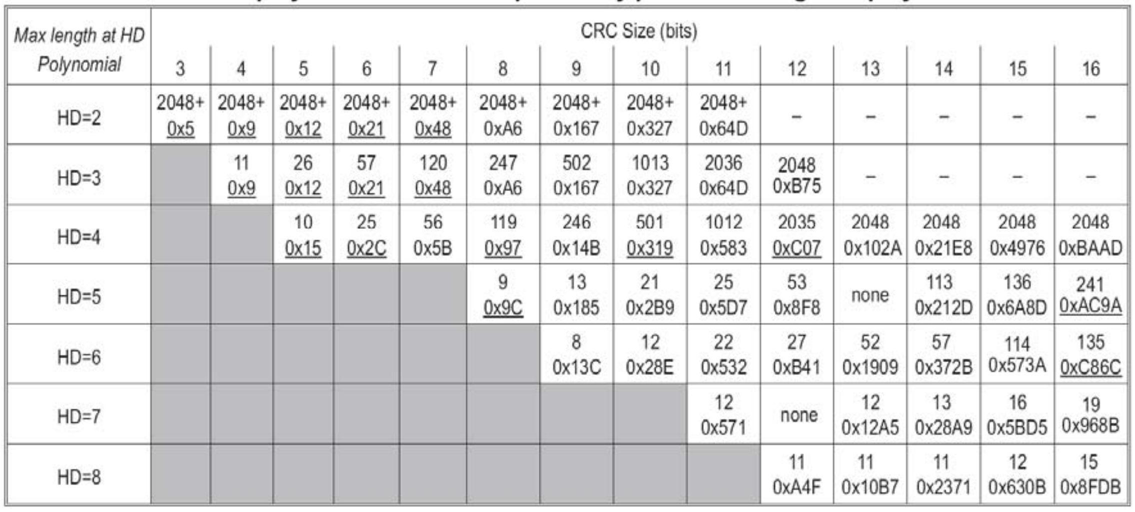


Source:
Koopman, P. & Chakravarty, T., "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks,“ DSN04, June 2004

Optimal Polynomials For Small CRCs

- P. Koopman, T. Chakravathy, “Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks”, The International Conference on Dependable Systems and Networks, DSN-2004.

Table 3. “Best” polynomials for HD at given CRC size and data word length. Underlined polynomials have been previously published as “good” polynomials.



Max length at HD Polynomial	CRC Size (bits)													
	3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	25 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

More On Picking A Good CRC

- Important to select CRC polynomial based on:
  - Data Word length
  - Desired HD
  - Desired CRC size

  Safety-critical applications commonly select ***HD=6*** at max message length
- Good values also known for 24-bit and 32-bit polynomials
  - IEEE 802.3 standard gives HD=6 up to 268-bit data words
  - But 0xBA0DC66B gives HD=6 up to 16,360-bit data words
    - Koopman, P., "32-bit cyclic redundancy codes for Internet applications," International Conference on Dependable Systems and Networks (DSN), Washington DC, July 2002
  - We're working on assembling these in a convenient format
- Be careful of published polynomials
  - Get them from refereed publications, not the web
  - Even then, double-check everything!
    - (We found a typo within the only published HD=6 polynomial value in an IEEE journal)
  - A one-bit difference can change great ➔ horrible
  - Mapping polynomial terms to feedback bits can be tricky

Are Checksums Or CRCs Better?

- Checksums can be slightly faster in software (this is usually overstated)
  - But tend to give far worse error performance
    - Most checksum folklore is based on comparing to a *bad* CRC or with *non-representative* fault types

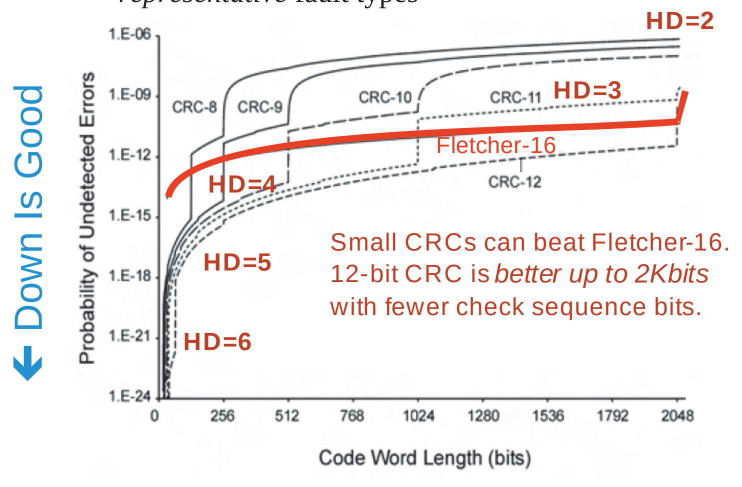


Fig. 12. Probability of undetected errors for Fletcher-16 and CRC bounds for different CRC widths at a BER of 10^{-5}. Data values for Fletcher-16 are the mean of 10 trials using random data.

Source:

Maxino, T., & Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, Jan-Mar 2009, pp. 59-72.

Aren’t Software CRCs Really Slow?

- Speedup techniques have been known for years
  - Important to compare best implementations, not slow ones
  - Some CPUs now have hardware support for CRC computation
- 256-word lookup table provides about 4x CRC speedup
  - Careful polynomial selection gives 256-byte table and ~8x speedup
  - Intermediate space/speedup approaches can also be used
  - Ray, J., & Koopman, P. "Efficient High Hamming Distance CRCs for Embedded Applications," DSN06, June 2006.
- In a system with cache memory, CRCs are probably not a lot more expensive than a checksum
  - Biggest part of execution time will be getting data bytes into cache!
  - We are working on a more definitive speed tradeoff study

Additional Checksum & CRC Tricks

- Use a “seed” value
  - Initialize Checksum or CRC register to other than zero
  - Prevents all-zero data word from resulting in all-zero check sequence
  - Can be used (with great care) to mitigate network masquerading
    - Transmitters with different seed values won’t “see” each others’ messages
- Be careful with bit ordering
  - CRCs provide burst error detection up to CRC size
  - Unless you get the order of bits wrong (as in Firewire)
  - Unless you put CRC at front instead of back of message
- CRC error performance is independent of data values
  - It is only the patterns of error bits that matter

Here There Be Dragons…

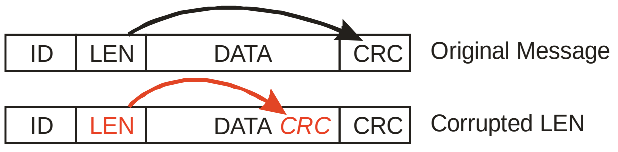
Other places to be wary (out of scope for our current research)

- Bit encoding interacts with CRCs
  - A one- or two-bit error can cascade into multiple bits as seen by the CRC
    - For example, bit stuffing errors can cascade to multi-bit errors
    - For example 8b10b encoding can cascade to multi-bit errors
  - Sometimes bit encoding can help (e.g., Manchester RZ encoding) by making it likely corruption will violate bit encoding rules

- Watch out for errors in intermediate stages
  - A study of Ethernet packets found errors happened in routers!
  - J. Stone and C. Partridge, “When the CRC and TCP Checksum Disagree,” Computer Comm. Rev., Proc. ACM SIGCOMM ’00, vol. 30, no. 4, pp. 309-319, Oct. 2000.

CAN vs. FlexRay Length Field Corruptions

- CAN does not protect length field
  - Corrupted length field will point to wrong location for CRC!
  - One bit error in length field circumvents HD=6 CRC



- FlexRay solves this with a header CRC to protect Length

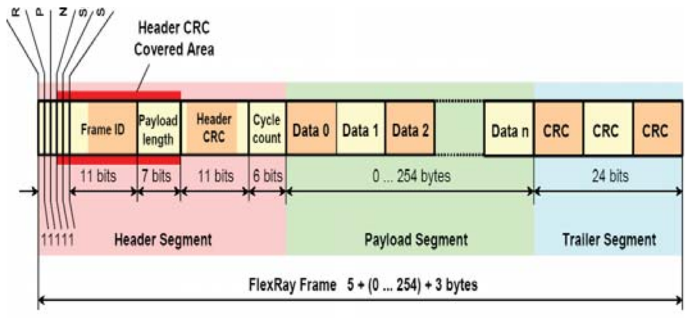
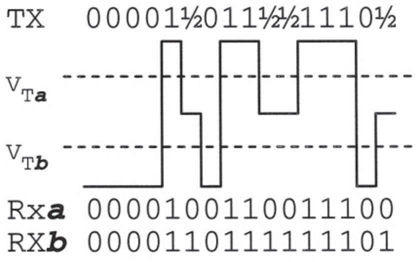


Figure 4-1: FlexRay frame format.

Source: FlexRay Standard, 2004

Byzantine CRC

- Byzantine failures for CRCs and Checksums



Example Schrodinger's CRC caused by non-saturated voltage values on a data bus. Two receivers (***a*** and ***b***) can see the same message as having two different values, and each view having a valid CRC

  - Paulitsch, Morris, Hall, Driscoll, Koopman & Latronico, "Coverage and Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," DSN05, June 2005.

- Memory errors may be complex and value-dependent
  – A cosmic ray strike may take out multiple bits in a pattern

Composite Checksum/CRC Schemes

- Idea: use a second error code to enhance error detection
  – Rail systems add a 32-bit "safety CRC"
  – Checksum + CRC can be a win ([Tran 1999] on CAN)
  – ATN-32 is a checksum used in context of network packet CRC
- Youssef et al. have a multi-CRC aviation proposal
  – Combines ideas such as "OK to miss an error if infrequent"
  – Uses composite CRCs based on factorization
  – Evaluated with random experiments
- Issue to consider:
  – What HD do you really get with a composite scheme?
    - E.g., which error patterns slip past both CRCs?
  – Are diverse checksum+CRC approaches better than dual CRC approaches?

Review

- Introduction
  - Motivation – why isn't this a solved problem?
  - Parity computations as an example
  - Error code construction and evaluation (without scary math)
  - Example using parity codes
- Checksums
  - What's a checksum?
  - Commonly used checksums and their performance
- Cyclic Redundancy Codes (CRCs)
  - What's a CRC?
  - Commonly used CRC approaches and their performance
- Don't blindly trust what you hear on this topic
  - A good CRC is almost always **much** better than a good Checksum
  - Many published (and popular) approaches are suboptimal or just plain wrong
  - There are some topics to be careful of because we don't know the answers
- Q&A

Investigators

- Philip Koopman Koopman@cmu.edu
  - Assoc. Prof of ECE, Carnegie Mellon University (PA)
  - Embedded systems research, emphasizing dependability & safety
  - Industry experience with transportation applications
- Kevin Driscoll Kevin.Driscoll@honeywell.com
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Ultra-dependable systems research & security
  - Extensive data communications experience for aviation
- Brendan Hall Brendan.Hall@honeywell.com
  - Engineering Fellow, Honeywell Laboratories (MN)
  - Fault tolerant system architectures & devlopment process
  - Extensive experience with aviation computing systems