PROPOSING IMPROVEMENTS TO AVOID THE CRC COMPROMISE AND THE SILENT SPECIFICATION

Cleon Rogers, Consultant, Little Rock, Arkansas

Abstract

In this paper we examine the issues and benefits associated with the use of cyclic redundancy checks (CRC)s that manifest during the later phases of safety-critical software development projects. We look at some less obvious details that will potentially increase return on investment (ROI), avoid common DO-178B certification planning and design difficulties, as well as, speed development and integration. We propose software build process improvements with promise to improve version integrity, executable integrity, version identification, and executable identification. We examine the role of CRCs in software configuration management (SCM) and propose their potential for partial satisfaction of DO-178B objectives, like traceability. We look at time-saving steps for integrating with systems that use one of three standard algorithms. Of particular interest, we provide a table of residues from faulty algorithms and their likely causes and remedies, and provide overlooked tips for asynchronous communication devices to aid in the late development phase. There are analysis and data collection techniques proposed to support initial data transfer error rate claims in the early assessment of system integrity and availability, and to show the satisfaction of established bounds.

Introduction

Successfully completing the development of embedded error detection and correction (EDC), and doing it faster, cheaper, and better is obviously possible when using experts [1,2,3]. To make the success more likely, in general, for future designs using these ubiquitous EDCs, like CRCs, is a goal of this paper, through the potential of improved standards, specifications, and guidance. The number of unsuccessful completions is unknown and, in the opinion of the author, their identification needs to be established. Likewise, a turn-key "checklist" type of solution could supplement existing data in specifications, thus reducing the effects of the "Silent Specification," (a specification missing the necessary details for successful completion without consulting subject matter experts in the late development phase). Better, more complete, specifications prevent short circuits to release from becoming the driving current to the "CRC Compromise." We define the CRC Compromise as a situation where the error control (or CRC) has been disabled, by either turning its calculation and comparison off, or by using a pre-calculated receiver-end, "actual" CRC value that will always match the transmitter-end "expected" value. With a silent specification and/or lacking standard guidance, this activity might pass through certification. We gain so much, and economically, from using the robustness of error control correctly, (we present some here), that these hazards should be, in our opinion, recognized and prevented by future guidance. As more functionality is moved from mechanical and electrical systems to embedded electronic and software systems, the protected cargo (embedded code) must be executed dependably.

The paper is organized by following the life cycle steps of an embedded code, which will be partially protected during runtime by an appended CRC (or multiple CRCs). Along the way, other potential benefits and issues of CRC usage are summarily noted, as they arise. Finally, we end with the protection and identification provided to the code in data storage archives of the configuration management versions and their retrieval for subsequent usage in any new generation developments. (*Please note that following the path of embedded code is for organizational purposes. The collection of CRCs for a complete system may include the ones covering source files, etc.*) We categorize the path into ten steps, some integral, for reference:

A Typical Journey of Our Embedded Code

Step 1: Specifications/Standards/Planning/Design/ Develop/Build/Make

Step 2: Version Control/Project Management

Step 3: Module/Interface Testing

978-1-4244-4078-8/09/$25.00 ©2009 IEEE.

Step 4: Release/CM/QA

Step 5: Formal Verification/Validation/ Certification

Step 6: Distribution/Loading/Runtime

Step 7: Historical Data Collection/ Maintenance/ Modifications

Step 8: Repeat steps 1 through 6 for modifications

Step 9: Archival/End-of-life

Step 10: New Generation Inclusion (Reuse)

Background

Background to the current subject matter was provided in our previous paper [4]. Whereas the previous paper focused on issues surrounding the field-loading of software, this paper expands on issues and benefits of error control, in a broader sense, to aid non-experts in their usage, and to raise awareness of the potential for improvements to standards and guidance.

Methods

For this paper, we collected and analyzed information from text books, research papers, the internet, lessons learned on previous projects, standards, regulations, as well as some that may be new, at least in the collection [5-12].

The Journey

Step 1.1 Specification

Proposed for inclusion in a specification, standard, or elsewhere are:

(a) require an appropriate error control strategy [1];

(b) consider the common mode for the CRC covering the executable code residing in a data storage archive of software versions in dual redundant systems [13];

(c) consider stating the message block size minimum, maximum, and size restrictions for the chosen polynomial; include the message length as a parameter to the calculation routine [14,15];

(d) establish restrictions on channel noise by setting bounds for bit error rates (BER);

(e) requiring compliance with improved standards covering error control;

(f) requiring research references and sourcing for chosen polynomials and implementations; or improved standards and guidance;

(g) restricting processor selections with instruction sets that affect CRC implementations, for instance, left/right shift and Exclusive OR;

(h) consider requiring hardware counters for all jumps through RESET, subsequent to the last scheduled maintenance reading and clearing of counters;

(i) consider requiring the collection of error statistics on data transfers, such as, retries, overruns, parity errors, underflow, framing errors, timeouts, length errors, etc.

(j) specifying protocol bit, byte, word ordering and other data used in the CRC calculation of a protocol packet;

(k) stating the endianess of the chosen processor and coordinating with the CRC algorithm implementation;

(l) include a runtime verification method of table entries of any table-driven algorithms; and

(m) several references provide additional standard-type parameters for CRC implementations, such as, stating the initial CRC value, final XOR value, forward or reverse algorithms, padding, flushing, word alignment, etc. [12, 10, 9].

Step 1.2 Standards

We will cover later, two 16-bit standards, CRC-16 and CCITT-16 and one 32-bit standard, CRC-32. It is recommended that standards be augmented with information on block size restrictions, their effects on the probability of undetected errors, and the patterns for detectable and undetectable errors (i.e., random and/or burst), for future designs.

Step 1.3 Planning

An inclusion to the basis of certification in the Plan for Software Aspects of Certification (PSAC) is proposed to reference the specification and an improved CRC Standard for agreement on the error control usage to partially satisfy identification and integrity requirements.

Step 1.4 Design/Develop

Using the information collected from the specification, standards, PSAC, etc. the EDC/CRC software or system developer should be able to select, reference, and specify one of the many available peer-reviewed, published, standards-based implementations that satisfy the requirements, that is unambiguous to implement, accurate, and that is verifiable. Tables, from table-driven implementations, and CRC algorithms come as matching sets, such as, a "forward" 16-bit table is coupled with a forward 16-bit algorithm, both using the same polynomial. One possible implementation hazard would use the 16-bit table from one algorithm with the calculation algorithm from another.

Step 1.5 Build/Make

In the 'make' of systems, utilizing table-driven algorithms, the table generator code could be exercised during every build. One way is to verify a match of the new table, when created, in the build with the table that was archived in the previous version. In addition, the source files used in the previous build could have their file CRC values recalculated and verified before applying the deltas to these files and using them in the new build. Periodically, older versions in SCM could have all CRCs recalculated and verified, also. For added assurance, run a pass of the executable with each byte fed to the algorithm bit-reversed and store the result in the version description document, along with the normal one. In addition, the forward and bit-reversed CRCs could be embedded in the executable code. It is stated here without proof that this two-pass method, with what we'll call Secondary CRC(s), could approach a doubling of the protection against undetected errors, with a negligible increase in complexity.

In the version description document or software configuration index, include the executable's CRC(s) with the version number. This provides a unique version identification and a traceability link from the executable to the version, used for partial satisfaction of DO-178B objectives. If this same ID information is read and displayed, it could provide executable identification, another DO-178B objective. The uniqueness of the version and executable identification improves integrity.

Step 1.6 Unit Test

Consider incremental development of a unit test, starting with a simple message, such as, 'T', for the byte-at-a-time methods using table lookups. The byte character, 'T', will result in an index fetch into the table with the addition of pre- and post-conditioning. Note that this index is bit reversed by some algorithms. Unit testing with word or longer messages should match machine byte and word ordering specifications of the implementation. Several practical references give step-by-step instructions with sample code for some of the standard polynomials. When developed, the verified results could be recorded against published peer-reviewed examples [16-19]. If it hasn't been done previously, messages and CRCs could be matched against a polynomial or synthetic division manually, or with the aid of a computer algebra system (see step 5).

Step 2 Version Control & Project Management

Consider controls to prevent unauthorized 'restores' from backups of the version control database that could lose newly identified hazards and safety discrepancies that are to be reported to program management and the developer.

Step 3 Module/Interface Testing

In Table 1[1], the first column, "Poly", contains the polynomial coefficients. Column 1 contains the residue (CRC) obtained using the character, 'T', (hexadecimal value, 0x54), with parameters: all zeros for the initial value; zeros for the final exclusive or (XOR); not bit reversed on input; and not bit reversed on output. Going from column 1 to column 2: reverse the bits in each input byte, reverse the bits in

[1] Data for table entries provided by Alex Rogers, (used with permission).

each output byte; from 2 to 3: each output byte is not reversed; from 3 to 4: use all ones for the initial value, each output byte is reversed; from 4 to 5: each input byte is reversed, each output byte is not reversed; from 6 to 7: each input byte is not reversed. If a match is not found in the expected column and is found in another column, follow the steps for the two columns, from left to right, and undo the incorrect steps in the code and/or data.

Table 1. Faulty Residues and Their Parameters

Poly	1	2	3	4	5	6	7
1021	1A71	14A1	8528	1B26	81DF	9B27	047E
8408	9AF8	1AB1	8D58	11BA	1452	A277	B5D7
A001	C06E	EC06	6037	61FC	FBF9	3F86	9FDF
8005	81FB	FF01	80FF	BFBE	9F3E	8202	8306

Step 4 Release/CM/QA

The data storage of software life-cycle data (SLCD) has not necessarily been adequately protected with a CRC that is designed for an ARQ system [1], and becomes a common hazard in the extraction of released executable code from storage and then installed on a dual-redundant system (see Figure 1). Any residual risk must be knowingly accepted, according to assessment practices of some system safety standards [20].

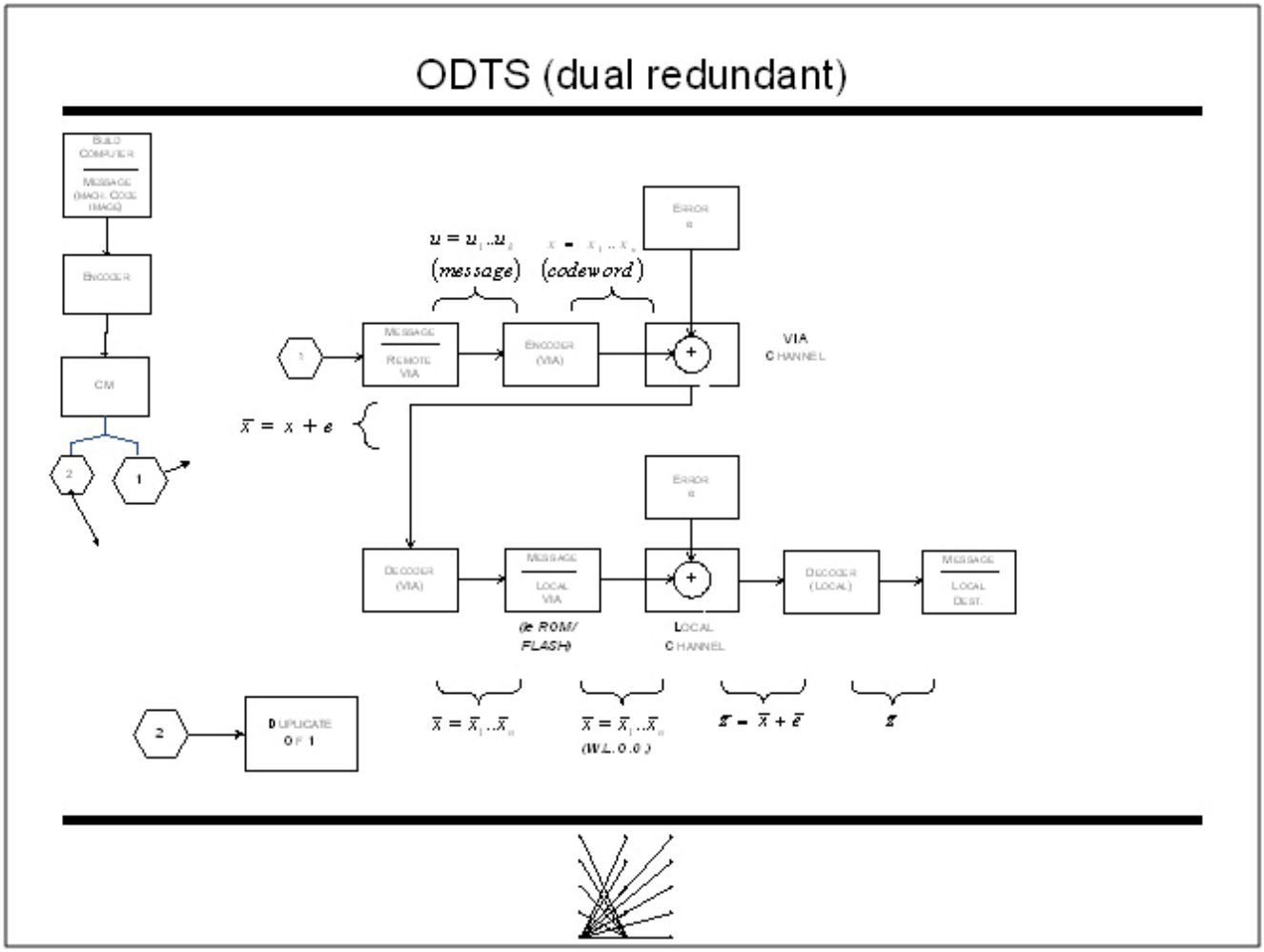

Figure 1. Overall Data Transfer System (ODTS) – Dual Redundant

It is proposed that the CM process associate a CRC (or EDC) with the version number included in the SLCD, such as, a version description document (VDD), equipment specification, or software configuration index (SCI). The version data should include the software life cycle environment source code data that is used for calculating and embedding CRCs. (see section 11.15 of Software Considerations in Airborne Systems and Equipment Certification) [21]. For table-driven algorithms, this source data should include the table, the source code that generates and verifies the table, and the mechanism that prevents the table entries from accidental and/or unauthorized alterations. It is recommended that the table generation be exercised for every release. Periodic recalculation of all CRCs supports the satisfaction of data recovery and retention objectives.

For a more critical embedded system, one could run another pass of the algorithm with a different polynomial and include the new CRC with the original; or run a pass with each of the bytes, bit-reversed, and store this CRC in addition to the original. (*This technique, using Secondary CRC(s), was first mentioned in step 1.5, but to serve a different primary purpose, integrity vs. identification.*)

Step 5 Formal V&V/Certification

Rules of thumb for inspection of the tables in byte-oriented table-driven methods:

1. The first table entry is generally all zeros.
2. The second entry is a repeat of the CRC polynomial coefficients.
3. For reversed methods, the CRC polynomial coefficient repeat is located at hex address 0x80.
4. Also note, for a one byte message equal to hex 0x01, the calculation should return the first table entry with pre- and post-conditions, and reflections applied.

Polynomial or synthetic division of the message by the CRC polynomial (generator polynomial) can be done manually to confirm the results of an algorithm on a given message:

Polynomial Division over a Field [22]

Let the message be m(x), the CRC polynomial be p(x), and the quotient is q(x) with the remainder of r(x), then say

$$m(x)=\sum_{i=0}^{k} c_i x^i ,$$

$$p(x)=\sum_{i=0}^{l} d_i x^i \text{ , over a field, } k \ge l \ge 0 ,$$

$$d_l \neq 0 ,$$

so

$$d_l x^l + d_{l-1}x^{l-1} + \cdots + d_0 x^0 \overline{)\, c_k x^k + c_{k-1}x^{k-1} + \cdots + c_0 x^0}$$

has

$$q(x)=\sum_{i=0}^{k-l} q_i x^i \text{ and}$$

$$r(x)=\sum_{i=0}^{l-1} r_i x^i$$

satisfying the Euclidean property over an integral domain,

$$m(x)=p(x)q(x)+r(x),$$
$$\deg[r(x)]<\deg[p(x)].$$

Step a:

$$For(n=k-l;n=0;n--)$$

$$q_n=\frac{c_{l+n}}{d_l}$$

$$For(j=l+n-1;j=n;j--)$$

$$c_j=c_j-q_n d_{j-n}$$

Step b:

$$return((q_i;i=0,1,\cdots,k-l),(r_i;i=0,1,\cdots,l-1))$$

from step a, where $r_i = c_i$.

The CRC to be appended to the message, m(x), is just the coefficients of r(x). (Note: For binary and monic cases, $d_l = 1$, and thus, the synthetic division is

performed using only subtraction and multiplication, no division. Also, returning q(x) is not necessary, it could be discarded.)

Provided below is a procedure using Mathematica that could ease the tediousness of the modulo-2 polynomial long division for a long message. The provided procedure is demonstrated with an example from Joe Campbell's book [9]:

```
<< Algebra`PolynomialPowerMod`


[ASCII Message char 'CfyU' w/bit 8 for EVEN parity, 0x1021 Divisor (11021)]


msgBcoef = {1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0,
    1, 1, 1, 1, 1, 0,
  0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0,
  0}


msgBinPoly = Apply[Plus, msgBcoef*Table[x^n, {n,
    47, 0, -1}]]


divisorCoef = {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 1}


divisorPoly = Apply[Plus, divisorCoef*Table[x^n,
    {n, 16, 0, -1}]]


remPolyIsCRC =
    PolynomialRemainder[msgBinPoly, divisorPoly,
    x, Modulus -> 2]


msgCcoefWithCRC =
  BitXor[msgBcoef,
   Join[Table[0, {n, 1, 32}],
    Reverse[CoefficientList[remPolyIsCRC, x]]]]


msgCWithCRCasPoly = Apply[Plus,
    msgCcoefWithCRC*Table[x^n, {n, 47, 0, -1}]]


zeroRemQ =
  PolynomialRemainder[msgCWithCRCasPoly,
    divisorPoly, x, Modulus -> 2]
```

In addition, a CRC procedure is provided[2] in the Python language:

```
    class Cyclic():
      def __init__(self):
        self.table = [None]*256
      def
crcInit(self,poly,order='forward'):
        for dividen in range(256):
          remainder = dividen << 8
            if order == 'forward':
              for bit in range(8):
                if  remainder  &  0x8000:  #
don't    skip col in long div (topbit)
                  remainder = (remainder <<
1) ^ poly
                else:  # skip col, divisor >
remainder, for this col
                  remainder = remainder << 1
              self.table[dividen]          =
remainder & 0xFFFF # keep 16 bits
            elif order == 'reverse':
              for bit in range(8):
                if  remainder  &  0x0001:  #
don't skip col in long div (botbit)
                  remainder = (remainder >>
1 ) ^ poly
                else: # skip col, divisor >
remainder, for this col
                  remainder = remainder >> 1
              self.table[dividen]          =
remainder & 0xFFFF # keep 16 bits
            else:
              print   'Use   "forward"   or
"reverse"'
              exit()
```

[2] Python code developed and provided by Alex Rogers, (used with permission).

```
      def
crcCalc(self,message,initVal=0x0000,
finalXOR=0x0000,bit_rev_in='n',bit_rev_out='
n',word_order='A'):
        for byte in message:
          if bit_rev_in == 'n':
            byte = ord(byte)
          elif bit_rev_in == 'y':
            byte = bit_rev8(ord(byte))
          else:
            print 'Invalid input. Please use
y/n'
          index = (( initVal >> 8 ) ^ byte )
& 0xFF # chg shift L/R
          crc    =    (self.table[index]    ^
(initVal << 8)) # chg shift L/R
        if bit_rev_out == 'n':
          return '%x' % ((crc ^ finalXOR) &
0xFFFF) # return 16 bit CRC
        elif bit_rev_out == 'y':
          return  '%x'  %  (bit_rev16((crc  ^
finalXOR) & 0xFFFF)) # return 16 bit rev CRC
    ---------------------------------------
    if __name__ == "__main__":
        a = Cyclic()
        a.crcInit(0x1021,order='forward')
        print
a.crcCalc('T',initVal=0x0000,finalXOR=0x0000
,bit_rev_in='n',bit_rev_out='n')
```

Step 6 Distribution/Loading/Runtime

Issues encountered in the loading of embedded code were covered in our previous paper.

During distribution, the released embedded code is exposed to data handling and transfer threats. These threats also exist during the release process and are described, later, in the discussion of step 4. Likewise, the EDC/CRC appended during the software build could be expected to cover these unprotected gaps in the distribution process.

During runtime, if the data protected by the CRC is exposed to 100% noise, then there is a 100% probability of failure [23].

During runtime, using an ARQ strategy in a noisy environment, the quantity of retries can cause an unacceptable impact on throughput [1]. For example, in a sufficiently noisy environment, using only an error detection strategy, a resetting system may never reboot, because it always fails the CRC comparison.

Step 7 Historical Data Collection/Maintenance/ Modifications

In step 6, there were cases where noise caused an unacceptable impact on the system. While it may be impossible or impractical to eliminate the risk due to noise in all cases, collecting historical data during runtime is a common practice for characterizing a channel. Asynchronous communications devices are designed to provide data transfer statistics to embedded software, such as, framing errors, overruns, underruns, parity errors, etc. One highly recommended reference, [9], makes a comment that these statistics are of little value, we disagree for safety-critical systems. The incorporation of the collection of these statistics into the system design and maintenance strategy could support claims of bounds on channel noise and provide early warnings of degrading hardware components to vehicle health maintenance systems.

Step 8 Repeating Steps 1 Through 6

When necessary modifications are discovered, steps one through six are re-entered with proper authorization and notifications, which should result in a new set of unique CRCs. If by the extremely remote chance it doesn't, a procedural step could be in place to make another modification that forces a unique set, before approving the release. In other words, check all the new CRC's uniqueness with respect to the previous ones before approving a release.

Step 9 Archival/End-of-life

DO-178B has data retention, data retrieval, and integrity control requirements that could be partially met by the uniqueness and protection provided by the CRC(s) traced to the versions. This practice may, already, be the de facto standard. File sizes are always increasing. The maximum block size that can be protected by a 32-bit CRC is 512MB. The maximum block size that an algorithm will maximally protect, in general, is a necessary

parameter to be included in standards, guidance, and specifications to assist developers and system verifiers in showing compliance with safety requirements.

Using an FEC strategy is recommended for the data storage of archives [1]. Data storage equipment typically uses an FEC strategy. We are recommending a separate FEC strategy to cover the gaps in data transference and elsewhere.

Step 10 New Generation Inclusion (Reuse)/OTS (Off-the-Shelf)

Satisfying the data retrieval requirements mentioned in step 9, apply to data retrieved for the purpose of reuse and/or off-the-shelf. Verifying and identifying the code with the CRC before reuse, or use as off-the-shelf, could be included in a guidance checklist.

Discussion of "The Journey"

We will now revisit and further discuss some of the benefits, issues, and proposed improvements of CRC usage that arose in the journey steps, enumerated in the previous section.

First, step 1.1 (a) proposes using an appropriate control strategy. At least two types of strategies exist, a FEC and a ARQ. FEC is recommended for data storage and some hybrid systems. ARQ is common in communications [24, 25]. The ARQ strategy assumes the existence of a feedback mechanism with the ability to retransmit until a reliably correct transfer has been completed. In long term data storage or in flight, the CRC over embedded code doesn't have a practical, suitably reliable feedback path. In a noisy environment, at 35,000 feet and 35° LAT, even a dual-redundant system could fail to reboot either side, after a RESET or failover, which employ only an error detection (ARQ) strategy.

In step 3, there was a table of residues (CRCs) provided, to assist during integration, of results from improperly implemented algorithms or from using different parameters.

In step 4, it was pointed out that a CRC could be used for both the improved identification and enhanced integrity of embedded code for its versioning. One of the hazards during the release process is the handling of data. For example, if data is transferred to a removable USB storage device and then the device is removed before a cached write has been completed, the data could be corrupted. There are warnings about this practice, but it is commonly ignored. For another example, if data is transferred on a network, the network probability of an undetected error may exceed safety requirements for critical software. The EDC/CRC appended during the software build could be expected to cover these unprotected gaps in a release process. The build appended CRC is the typical risk mitigation measure used for these scenarios.

In step 5, we saw some rules of thumb that could assist in the verification of table-driven algorithms, by inspection. In addition, we saw that there exist alternate methods of calculating the CRC of a given message. Three alternates were provided, synthetic division, symbolic polynomial long division in a Mathematic procedure, and a Python language procedure.

In step 8, a procedure was described for handling the situation of a build of two different versions resulting in identical CRC(s). Most often, when the CRC(s) are the same for two subsequent versions, it indicates that the CRC(s) are not actually being calculated over new versions of the embedded code, but rather, they are being 'hard-coded' into the code. Again, in our opinion, this is an unacceptable practice, "The CRC Compromise."

Conclusions

While it may be impossible or impractical to design a completely hazard-free system, any remaining threat from residual hazards must be acceptable and acknowledged [20].

Error control safety analysis, development, and verification are different from the same done for either hardware or software, in some respects. First, hardware has a long established means for predicting the probabilities of component and system failures, but software doesn't. So for software, DO-178B is considered as a means of acceptance of software systems by regulators, and with good results. DO-178B is a design assurance based on controlling the process of software development. However, software algorithms for error control, which currently fall under DO-178B, are between the two, hardware and

software. Probabilities for EDCs can be analyzed and calculated outside the software development process, and thus might be shown to satisfy regulations at a system level by an alternate means, assuming compliance with some future established guidance.

As we have seen, the EDC/CRC is called upon to identify and protect from threats and hazards, encountered along the journey, its embedded code cargo, from the build to end-of-life, and then again, possibly, in reuse. Its journey could literally be thousands or millions of miles in distance and decades in time, exposed to failures in strategy, storage retrieval, uncontrolled noisy environments, mistaken identity, unprotected gaps in transference, write caching errors, automatic translations, operating system timing, etc.

As a conclusion, it is advocated:

- studies be conducted for determining and defining the size and scope of unsuccessful completions versus successful completions;
- collect best practices from advanced industries (i.e., disk, processor, and communications);
- research and publish best practices and suggest remedies for unsuccessful ones;
- incorporate the results in the development and adoption of supplements for governing documents that provide guidance, standards, and/or specification information for safety assessment, system design and development;
- establish metrics for their usage by reference;
- coordinate theses activities, as appropriate, within the industry and government;
- disseminate the conclusions to the public;
- follow up on their impact and return on investment; and
- provide assistance with the technology transfer to other interested industries.

References

[1] Lin, Shu, Daniel J. Costello, Jr, 1983, Error Control Coding: Fundamentals and Applications, Englewood Cliffs, New Jersey, Prentice-Hall, Inc.

[2] Wolf, Jack Keil, Robert D. Blakeney, II, 1-Sep-88, An Exact Evaluation Of The Probability Of Undetected Error For Certain Shortened Binary CRC Codes, IEEE Transactions On Communications, pp. 287-292.

[3] Berlekamp, Elwyn R., 1968, Algebraic Coding Theory, McGraw Hill, Inc.

[4] Rogers, Cleon, 29-Oct-08, Choosing A CRC & Specifying Its Requirements For Field-Loadable Software, 27th Digital Avionics Systems Conference, Minneapolis, MN.

[5] Fujiwara, Tohru, Tadao Kasami, Atsushi Kital, Shu Lin, 1-Jun-85, On the Undetected Error Probability for Shortened Hamming Codes, IEEE Transactions On Communications, Vol Com-33 No 6, pp. 570-574.

[6] Peterson, W.W., D.T. Brown, 1-Jan-61, Cyclic Codes For Error Detection, Proceedings of the IRE, pp. 228-235.

[7] Wicker, Stephen B., 1995, Error Control Systems for Digital Communication and Storage, Upper Saddle River, New Jersey, Prentice-Hall, Inc.

[8] Perez, Aram, 1-Jun-83, Byte-wise CRC Calculations, IEEE Micro, pp. 40-50.

[9] Campbell, Joe, 1987, C Programmer's Guide to Serial Communications, Carmel, Indiana, Macmillan, Inc.

[10] Nelson, Mark, 1992, Serial Communications A C++ Developer's Guide, USA, M&T Publishing, Inc.

[11] Schwaderer, W. David, 1992, C Programmer's Guide to NetBIOS, IPX, and SPX, Carmel, Indiana, Howard W. Sams & Company.

[12] Williams, Ross, 1993, A Painless Guide to CRC Error Control Codes, www.ross.net/crc.

[13] Storey, Neil, 1996, Safety-Critical Computer Systems, Essex, England, Addison Wesley Longman Ltd.

[14] Stone, Jonathan, Michael Greenwald, Craig Partridge, James Hughes, 1-Oct-98, Performance of Checksums and CRC's over Real Data, IEEE/ACM

Transactions on Networking, Vol 6 No 5, pp. 529-543.

[15] MacWilliams, F. J., N. J. A. Sloane, 2006, The Theory of Error-Correcting Codes, 12th Ed., Amsterdam, The Netherlands, Elsevier B. V.

[16] Leung, C., K. A. Witzke, 1-Dec-90, On Testing for Improper Error Detection Codes, IEEE Transactions On Communications, Vol 38 No 12, pp. 2085-2086.

[17] Ramabadran, Tenkasi V., Sunil S. Gaitonde, 1-Aug-88, A Tutorial on CRC Computations, IEEE Micro, pp. 62-74.

[18] Shouse, D. V., 1-Apr-85, "On the Fly" CRC-16 Byte-wise Calculation for 8088-based Computers, IEEE Micro, pp. 67-75.

[19] Witzke, K. A., C. Leung, 1-Sep-85, A Comparison of Some Error Detecting CRC Code Standards, IEEE Transactions On Communications, Vol Com-33 NO 9, pp. 996-998.

[20] MIL-STD-882D, 2000, Standard Practice for System Safety, HQ AFMC/SES, Wright Patterson AFB, OH.

[21] Software Considerations in Airborne Systems and Equipment Certification, 1992, SC-167, Washington, DC, RTCA, Inc.

[22] Akritas, A.G., 1989, Elements of Computer Algebra with Applications, John Wiley & Sons, Inc.

[23] Shannon, C.E., 1948, A Mathematical Theory of Communication, The Bell System Technical Journal, Vol 27, pp. 379-423, 623-656.

[24] IEEE Std 802.3-2005, 2005, IEEE, pg 52.

[25] Hammond, J.L., J.E. Brown, S.S. Liu, 1975, Development of a Transmission Error Model and an Error Control Model, RADC-TR-75-138, Griffiss AFB, NY, RADC (RBC).

Email Addresses

Cleon Rogers: rogers@swbell.net
or cleonrogers@gmail.com

28th Digital Avionics Systems Conference
October 25-29, 2009