ZESZYTY NAUKOWE POLITECHNIKI RZESZOWSKIEJ 288, Mechanika 85
RUTMech, t. XXX, z. 85 (4/13), październik-grudzień 2013, s. 421-430

Krzysztof GŁOWACKI[1]

# AVIONICS SYSTEMS SOFTWARE DEVELOPMENT ACCORDING TO THE METHODOLOGIES CONFORMING DO-178B

Avionics systems software of modern aircraft must fulfil rigorous requirements of reliability because of executing critical tasks which have a direct impact on flight safety (safety critical software). Development of such software is a tremendous project which main goal is to produce software according to the methodology conforming DO-178B guidelines. The paper answers the question how generally known and widely used standards for software development influence the reliability of software being developed. Special emphasis was placed on the coding phase (development of the source code). The paper presents examples of software bugs that result from lack of adherence (or violation) to rules and guidelines of standards. It discusses their impact on the software and generally on the system, and their consequences. Basing on own experience, the author convinces that obeying the standards not only unifies both documentation and the source code making them more readable and maintainable but also, what is of key importance from reliability point of view, prevents producing software bugs.

**Keywords:** DO-178 standard, avionics systems software, flight safety

## 1. Introduction

Airborne systems of modern aircraft must fulfil rigorous requirements of reliability because of executing critical tasks which have a direct impact on flight safety (*safety critical*). Software reliability can be achieved by following three concepts during software development:

- error avoidance (by following safe practices),
- use of fault tolerance mechanisms (e.g. redundancy, diagnostics, BIT),
- testing (error detection and elimination).

Complete error avoidance in the software is impossible. Therefore there is an absolute necessity to minimize the probability of its occurrence. This is the main goal of DO-178B standard [1] that delivers guidelines for software development. Confirming these guidelines assures developing and employing a highly reliable

[1] Autor do korespondencji/corresponding author: Krzysztof Głowacki, MTU Aero Engines Polska, Tajęcina 108, 36-002 Jasionka, tel. (17) 7710482, e-mail: Krzysztof.GLOWACKI@mtupolska.com

software development methodology. Guidelines of DO-178B standard are used widely in projects provided by MTU Aero Engines Polska. Software Department of Engine Control and Monitoring Systems from its very beginning takes part within e.g. the following software development and certification projects presented in Table 1.

Table 1. Software development and certification projects

| Aircraft | Project | Unit |
|---|---|---|
| Eurocopter Tiger | MTR390 ECMU | Engine Control and Monitoring Unit |
| Airbus A400M | TP400 ECU | Engine Control Unit |
|  | TP400 EPMU | Engine Protection and Monitoring Unit |
| Eurofighter Typhoon | EJ200 DECMU | Digital Engine Control and Monitoring Unit |

## 2. DO-178B in practice

DO-178B standard divides the area of software production into three key processes: SW Planning Process, SW Development Process and Correctness Process. The last one includes all activities that ensure correctness and quality of software. According to DO-178B standard, effective planning is a determining factor in producing software. The software life cycle, the software development methodology and its model are defined during the planning phase together with methods and tools that will be used to produce the software as well as specific plans and standards for each phase of software development. It is worth noting that these plans and standards are developed for the need of particular project (therefore they are called as project plans and project standards). This is required by the certification process. Project plans and standards constitute the basis for certification, it means that certification authority at first approves them, and then checks if the software development was conducted according to the project plans and verifies the adherence of the software and associated documentation to the project standards. Considering SW Development Process, DO-178B standard distinguishes its 3 stages, for all of them it requires correspondingly three project standards:

- Software Requirements Process,
- Software Requirements Standard (SRSTD),
- Software Design Process,
- Software Design Standard (SDSTD),
- Software Coding Process,
- Software Coding Standard (SCSTD).

Project standards used in different avionics software projects are similar. The differences can result from establishing different methods and practices of software development, different tools or programming languages used, and finally different hardware platforms. Similarities come first of all from the fact that most likely almost all project standards are developed on the basis of well-known and widely used software standards. There are numerous popular and acclaimed guidelines and standards for software development. It is worth mentioning as examples standards of programming languages (C, C++, Ada) published by such organisations as ANSI, IEEE, IOC or IEC. Additionally, there is a full range of standards for integer and floating point arithmetic, interface standards (ABI, EABI, API, ASI), and many others. All these, if they are helpful in particular project, can be used as a basis for development of project standards.

## 3. Standards

Every software engineer is aware that low quality requirements can lead to their incorrect interpretation, and in consequence to their incorrect implementation in the software. To prevent this kind of mistakes rules and guidelines of standards dedicated for software requirements have to be followed. Below there are of key importance attributes of high quality requirements according to ANSI/IEEE 830-1998 standard [2]: correctness, unambiguousness, completeness, coherence, verifiability, modifiability, traceability. Standards dedicated for software architecture contain the architecture design concepts, methods, principles and practices of modelling and representing software architecture. The most popular concept widely used especially in large projects is a functional decomposition method according which system is broken down into parts that are easier to conceive, understand, program and maintain. The paper presents the examples of principles and rules that need to be followed in the modern airborne systems software: modularity, encapsulation, loose coupling, partitioning, portability, reusability. As an example of the known architecture standard that is worth mentioning, is ANSI/IEEE 1471-2000 standard [3]. Software coding standards (coding conventions) are a set of guidelines and rules for a specific programming language that recommend programming style, practices and methods that used to improve the readability of the source code, to make the software maintenance easier and to avoid making errors (producing software bugs). Coding standard rules cover such areas like programming style (e.g. indentation, nesting, capitalization), naming conventions (style of user-defined identifiers names, such as functions, types, variables, files), comment conventions (style of code comments, documenting changes, descriptions of algorithms) and programming constructs (programming language dependent rules that include both, recommended and forbidden structures that results e.g. from constraints of target hardware platform or development tools). From the software reliability point of view, the most important are rules connected to recommended and forbidden programming constructs [4-8]. Below a few of them are mentioned as examples:

- avoid hazardous techniques (some of them are useful but require great care)
  - use of *go to* statement,
  - use of pointers and pointer arithmetic,
  - use of parallel computing,
  - use of floating point arithmetic,
  - handling of interrupts and exceptions,
  - use of recursion,
  - use of dynamic memory allocation,
- apply safe techniques (*defensive programming*)
  - data protection techniques (modularity, encapsulation, memory management),
  - check for null pointers,
  - check for data validity (checksums, consistency checks, range checks),
  - check for response time (prevents blocking of the system),
  - check for divisions by 0,
  - check for overflows/underflows,
  - initialisation of all variables to safe values,
  - care about special cases and boundary conditions (empty sets, infinitive loops).

## 4. Examples

### 4.1. Time conversion (Ada95)

The programmer's task was to implement the following software requirement into the source code in Ada95 programming language: „Convert time read from RTC (Real Time Clock) device into seconds". Time read from RTC device was stored within the following variable:

```
-- Time read from RTC (Real Time Clock)
rtc_time : RTC_TIME_T;
```

Variable type was defined as follows:

```
type RTC_TIME_T is
   record
      HOURS   : U_INT_8; -- Possible values in range 0..23
      MINUTES : U_INT_8; -- Possible values in range 0..59
      SECONDS : U_INT_8; -- Possible values in range 0..59
   end record;
```

Calculated time in seconds was stored in a variable defined as follows:

```
-- Time as seconds
time_as_seconds : S_INT_16;
```

The Table 2 presents types of variables used within this paper and their ranges.

Table 2. Typer of variables and their ranges

| Type | Purpose | Range | Range |
|---|---|---|---|
| S_INT_8 | Signed Integer 8 Bit | $-2^7 \div +2^7 - 1$ | –128÷127 |
| S_INT_16 | Signed Integer 16 Bit | $-2^{15} \div +2^{15} - 1$ | –32768÷32767 |
| S_INT_32 | Signed Integer 32 Bit | $-2^{31} \div +2^{31} - 1$ | –2147483648÷2147483647 |
| U_INT_8 | Unsigned Integer 8 Bit | $0 \div +2^8 - 1$ | 0÷255 |
| U_INT_16 | Unsigned Integer 16 Bit | $0 \div +2^{16} - 1$ | 0÷65535 |
| U_INT_32 | Unsigned Integer 32 Bit | $0 \div +2^{32} - 1$ | 0÷4294967295 |

The programmer wrote the following lines of code:

```
-- Calculate time in seconds
time_as_seconds := S_INT_16(rtc_time.HOURS   * 3600) +
                   S_INT_16(rtc_time.MINUTES *  60 ) +
                   S_INT_16(rtc_time.SECONDS       );
```

During execution of tests on the software it has been observed that indications of time in seconds are correct only at the beginning of software execution. It is observed that approximately after 5 minutes time in seconds starts to indicate incorrect and random values. Analysis confirmed that the Source Code presented above contains software bugs. Compiler did not generate any error or warning (Green Hills Software Ada 95 Compiler). Analyses of the source code have been started in search of software bug from the point of view of possible overflows.

```
-- Sum of maximum values of each element can not be greater than
-- maximum value of time_as_seconds range
time_as_seconds :=
   S_INT_16(rtc_time.HOURS   * 3600) + -- max = 23 * 3600 = 82800 | > 32767
   S_INT_16(rtc_time.MINUTES * 60 ) + -- max = 59 *   60 =  3540 |
   S_INT_16(rtc_time.SECONDS       ); -- max = 59        =    59 |
                                       -- =======================|=========
                                       --            SUMA = 86399 | > 32767
```

The results were the following:

- overflow of `time_as_seconds` (its maximum value 86399 is not contained in range `S_INT_16`),
- overflow of `rtc_time.HOURS * 3600`.

(Its maximum value 82800 is not contained in range `S_INT_16`).

It can be easily calculated that the detected overflows will cause incorrect software behaviour after 9 h, 6 min and 7 s. During further analysis stated the following facts:

- compiler considers multipliers (values 60 and 3600) to be also `U_INT_8` type, because only then it can multiply elements `HOURS` and `MINUTES` by these factors,

- (note: type checking is specific for Ada programming language, operation of multiplication, addition, etc. are allowed only if data is of the same type, otherwise compiler generates an error and preclude software build),
- the result of these multiplications is stored as `U_INT_8`.

The following conclusions were reached:

- overflow of value 3600 (value 3600 is not contained in range `U_INT_8`)

  ```
  3600    : 00001110 00010000
  255     : 00000000 11111111
  ```

  Compiler considers the value 3600 as `U_INT_8`. So it cuts the most significant byte of value 3600 represented in binary format leaving the least significant byte. The value that was left in the least significant byte in decimal format is equal to 16. Therefore the conclusion is that the element `HOURS` is not multiplied by 3600, but always by 16,
- overflow of `rtc_time.MINUTES * 60` (its maximum value 3540 is not contained in range `U_INT_8`)

  ```
  3540    : 00001101 11010100
  255     : 00000000 11111111
  ```

The overflow occurs when element `MINUTES` reaches value 5.

The general conclusion is that the detected errors will cause incorrect software behaviour after 4 min and 59 s. The following solution was presented. First of all, the range of `time_as_seconds` should be increased (its type was changed from `S_INT_16` to `S_INT_32`). To avoid overflow of factor 3600, and products `rtc_time.HOURS * 3600` and `rtc_time.MINUTES * 60`, the type casting order was changed. Elements `HOURS` and `MINUTES` should be type casted to `S_INT_32` first, then multiplied by values 3600 and 60 (note that the factors will be considered by the compiler in this case to be `S_INT_32` type).

```
-- Calculate time in seconds
time_as_seconds := M_S_INT_32(rtc_time.HOURS  ) * 3600 +
                   M_S_INT_32(rtc_time.MINUTES) * 60   +
                   M_S_INT_32(rtc_time.SECONDS);
```

The source code above is correct, however it does not follow the following rule:

**„Untyped constants (named numbers) shall not be used.”**

The rule says that values which have specified meaning in the software should be represented by constants with defined type. Exceptions can be e.g.: value 0 (initialisation value), value 1 (incrementing, decrementing). The solution following the rule above is:

```
-- Calculate time in seconds
time_as_seconds := S_INT_32(rtc_time.HOURS  ) * HOUR_TO_SEC_CS   +
                   S_INT_32(rtc_time.MINUTES) * MINUTE_TO_SEC_CS +
                   S_INT_32(rtc_time.SECONDS);
```

where constants should be defined as follows:

```
-- Constant converting minutes to seconds
MINUTE_TO_SEC_CS : constant S_INT_32 := 60;
-- Constant converting hours to seconds
HOUR_TO_SEC_CS   : constant S_INT_32 := 3600;
```

Note that if the programmer follows the rule presented above writing this part of code, compiler would generate an error and preclude software build. Then the programmer would need to provide type casting taking full responsibility for correctness of these operations.

## 4.2. Boolean type variables (C)

There are three basic versions of C programming language known as ANSI C / C89 / C90, C99 and C11 defined by corresponding standards published by organisations ANSI, ISO and IEC. All three versions are very similar, however there are some differences. On the basis of C89/C90 and C99 comparison this example shows the difference of the usage of Boolean type and discusses misunderstandings resulting from this difference that can lead to producing software errors. Within C99 there is a built-in Boolean type. This means that including proper header (in this case `<stdbool.h>`) the programmer can use it. Within C90 there is no built-in Boolean type. That means it must be defined by the programmer manually (e.g. as a typedef or enum, as presented below).

```
/* Boolean definition by typedef */
typedef int boolean;
#define TRUE  1
#define FALSE 0
/* Boolean definition by typedef */
typedef int boolean;
#define TRUE  (1==1)
#define FALSE (!TRUE)
/* Boolean definition by enum */
typedef enum { FALSE, TRUE } boolean;
```

All three Boolean definitions presented above work in a program equally however in a different manner than the C99 Boolean type. The difference is shown by the following programming experiment.

There have been two very similar programs written and separately built by using the compiler GCC 4.5.3 (GNU Compiler Collection) configured for two versions of C programming language, corresponding C90 and C99. In both programs three Boolean variables have been created (Table 3). The task of both programs was to display the logical values of those variables and the results of conjunctions between them (Table 4). Conclusions from this programming experiment are presented in Table 5.

Table 3. Boolean variables

| Standard C90 | Standard C99 |
|---|---|
| `boolean b1 = FALSE;`<br>`boolean b2 = TRUE;`<br>`boolean b3 = 8;` | `bool b1 = FALSE;`<br>`bool b2 = TRUE;`<br>`bool b3 = 8;` |

Table 4. Logical values of Boolean variables

| Standard C90 | Standard C99 |
|---|---|
| `b1 : 0      b2 : 1      b3 : 8` | `b1 : 0      b2 : 1      b3 : 1` |
| `b1 && b2 : 0` | `b1 && b2 : 0` |
| `b1 && b3 : 0` | `b1 && b3 : 0` |
| `b2 && b3 : 0` | `b2 && b3 : 1` |
| `b2 && 8  : 0` | `b2 && 8  : 0` |
| `b2 && (boolean)8 : 0` | `b2 && (bool)8 : 1` |

Table 5. Conclusions from this programming experiment

| Standard C90 | Standard C99 |
|---|---|
| Operands have logical values:<br>- 0 (`FALSE`) when operand is equal to 0<br>- 1 (`TRUE`) when operand is equal to 1<br>- undefined when operand is not equal to 0 and not equal to 1 | Operands have logical values:<br>- 0 (`FALSE`) when operand is equal to 0<br>- 1 (`TRUE`) when operand is not equal to 0 |
| boolean b3 = 8<br>is not `TRUE` and not `FALSE` | bool b3 = 8<br>is `TRUE` |

Programmers using C99 version are safe but those coding in C90 that do not know the difference discussed above can produce software errors like this one presented below. The statuses of temperature measurement are read from the following buffer elements:

```
rx_buffer[4] /*   Sensor status (Bool)  U_INT_8  */
rx_buffer[5] /*   Signal status (Bool)  U_INT_8  */
```

It is assumed that buffer elements 4 and 5 contain statuses of correspondingly sensor and signal failures, where value 0 indicates failure occurrence, value 1 lack of failure. If at least one failure is detected the program should set the global temperature measurement flag to `TRUE`, otherwise to `FALSE`. The programmer wrote the following lines of code:

```
/* Set the temperature measurement failure flag */
if (rx_buffer[4] == 1) || (rx_buffer[5] == 1)
{  temp_measure_fail = TRUE;  }
else
{  temp_measure_fail = FALSE; }
```

The source code presented above is not correct due to the fact that sensor and signal statuses are stored in buffer element of type `U_INT_8` (possible value in range 0÷255). Therefore, in case of data corruption or transmission failure there is a possibility that buffer element 4 and/or 5 contain the value other than 0 and 1. In such case the program will set the flag `temp_measure_fail` to `FALSE` indicating lack of sensor and signal failures. This can lead to confidence that the temperature measurement and its value are correct. The programmer produces a software bug because he did not follow the following rule:

**„Boolean data shall be tested against 0 value (FALSE)."**

According to this rule the software requirement should be implemented in one of the following two correct ways:

```
        if (rx_buffer[4] == 0) && (rx_buffer[5] == 0)
        {  temp_measure_fail = FALSE;   }
        else
        {  temp_measure_fail = TRUE; }
```

or

```
        if (rx_buffer[4] != 0) || (rx_buffer[5] != 0)
        {  temp_measure_fail = TRUE;   }
        else
        {  temp_measure_fail = FALSE; }
```

It is worth a mention that the source code built by using compiler configured for C99 version of C programming language would generate correct program (`temp_measure_fail` would be set to `TRUE`).

## 5. Summary

The main goal of software production according to DO-178B standard is to increase the software reliability by avoiding errors during its development. For that purpose there is a necessity to strictly follow the rules and guidelines defined in standards. The examples presented within this paper show that following these rules prevent producing software bugs that can be undetectable during software build, and hidden or latent errors that reveal after some time of software run what is especially dangerous for airborne systems software. Conforming the rules and guidelines in the end saves time and effort needed for analysis of software requirements and source code in order to find and eliminate software bugs.

## References

[1] DO-178B: Software Considerations in Airborne Systems and Equipment Certification.

[2] ANSI/IEEE 830-1998: Recommended Practice for Software Requirements Specifications.

[3] ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems.

[4] MISRA-C:2004: Guidelines for the use of the C language in critical systems.

[5] ARINC Specification 653P1-2: Avionic Application Software Standard Interface.

[6] Hilderman V., Baghai T.: Avionics certification. A complete guide to DO-178 (Software), DO-254 (Hardware). Avionics Communications Inc., 2011.

[7] Maguire S.: Writing solid code: Microsoft's techniques for developing bug-free C programs. Microsoft Press, 1993.

[8] Dąbrowski W., Subieta K.: Podstawy inżynierii oprogramowania. Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Warszawa 2005.

*Projekt współfinansowany ze środków Unii Europejskiej z Europejskiego Funduszu Rozwoju Regionalnego w ramach Programu Operacyjnego Innowacyjna Gospodarka.*

## PROJEKTOWANIE OPROGRAMOWANIA SYSTEMÓW AWIONIKI ZGODNEGO Z WYTYCZNYMI STANDARDU DO-178B

**Streszczenie**

Oprogramowanie systemów awioniki współczesnych statków powietrznych ze względu na krytyczność wykonywanych zadań mających bezpośredni wpływ na bezpieczeństwo lotu musi spełniać zaostrzone kryteria niezawodności. Projektowanie takowych systemów bywa ogromnym przedsięwzięciem, którego celem jest wytworzenie oprogramowania zgodnie z metodologią spełniającą wytyczne standardu DO-178B. Niniejsza praca odpowiada na pytanie, w jaki sposób powszechnie znane i stosowane standardy wpływają na niezawodność projektowanego oprogramowania. Szczególny nacisk położono na etap kodowania (tworzenia kodu źródłowego). Na przykładach przedstawiono błędy w oprogramowaniu, które wynikają bezpośrednio z niestosowania się do reguł standardów, omówiono ich genezę, a także wpływ na działanie oprogramowania i całego systemu, oraz ich konsekwencje. Autor na podstawie własnych doświadczeń przekonuje, że stosowanie standardów nie tylko ujednolica i wprowadza przejrzystość zarówno w dokumentacji projektowej, jak i w kodzie źródłowym, ale również, co z punktu widzenia niezawodności systemów awioniki jest kluczowe, zapobiega popełnianiu błędów w oprogramowaniu.

**Słowa kluczowe:** oprogramowanie systemów awioniki, bezpieczeństwo lotu, standard DO-178B

DOI:10.7862/rm.2013.38

*Otrzymano/received: 25.10.2013 r.*
*Zaakceptowano/accepted: 22.11.2013 r.*