INTRODUCTION

This application note describes the Class B Safety Software Library routines that detect the occurrences of Faults in a single channel MCU. These routines have been developed in accordance with the IEC 60730 standard to support the Class B certification process. The routines can be directly integrated with the end user’s application to test and verify the critical functionalities of a controller without affecting the end user’s application.

This application note also describes the Application Programming Interface (API) functions that are available in the Class B Safety Software Library. The Class B safety software routines can be called periodically at start-up or run time to test the following components:

- CPU Registers
- CPU Program Counter
- Invariable Memory
- Variable Memory
- Clock

This application note also outlines various techniques, which are not part of the Class B Safety Software Library, to test components such as external communication, timing, I/O periphery, analog I/O and analog multiplexer.

OVERVIEW OF THE IEC 60730 STANDARD

The IEC 60730 standard defines the test and diagnostic methods that ensure the safe operation of the controlled equipment used in household appliances. Annex H of the IEC 60730 standard classifies the software into the following categories (see Appendix B: “IEC 60730-1 Table H.1 (H.11.12.7 of edition 3)“):

- Class A
- Class B
- Class C

The Class B Safety Software Library implements the important test and diagnostic methods that fall into the Class B category. These methods use various measures to detect and respond to the software-related Faults and errors.

According to the IEC 60730 standard, the controls with functions that fall into the Class B category should have one of the following structures:

- Single Channel with Functional Test
  In this structure, the Functional test is executed prior to the application firmware execution.

- Single Channel with Periodic Self-Test
  In this structure, the Periodic tests are embedded within the firmware, and the self-test occurs periodically while the firmware is in Execution mode.

- Dual Channel without Comparison
  In this structure, two independent methods execute the specified operations.

Note: The term ‘IEC 60730 standard’ that is used in this document refers to the “IEC 60730-1 ed.5.0” Copyright © 2013 IEC, Geneva, Switzerland. www.iec.ch.
SYSTEM REQUIREMENTS

The following system requirement is recommended to run the Class B Safety Software Library:

For the tests that require the independent time slot monitoring, the system hardware must be provided with at least two independent clock sources (e.g., internal oscillator, crystal oscillator and line frequency).

CLASS B SAFETY SOFTWARE LIBRARY

The 16-bit Class B Safety Software Library includes APIs, which are intended to maximize application reliability through Fault detection. These APIs help meet the IEC 60730 standard compliance. The following tests can be implemented using this library:

- CPU Register Test
- Program Counter Test
- Invariable Memory (Flash/EEPROM) Test
- Variable Memory Test
- Clock Test

In the following sections, the test description and the implementation details are discussed for each test. In addition, each section lists the APIs that are required to execute the corresponding test.

CPU Register Test

The CPU Register test implements functional test H.2.16.5, as defined by the IEC 60730 standard. It detects stuck-at Faults in the CPU registers. This ensures that the bits in the registers are not stuck at a value of ‘0’ or ‘1’.

The CPU Register test is a non-destructive test. This test performs the following major tasks:

1. The contents of the CPU registers to be tested are saved and restored during the test.
2. The registers are tested by first successively writing the binary sequences (length is dependent upon architecture), 010101... followed by 101010... into the registers, and then reading the values from these registers for verification.
3. The test returns an error code if the returned values do not match.

API FUNCTIONS

This API function implements the CPU Register test:

CLASSB_CPURegistersTest

Program Counter Test

The Program Counter (PC) test implements the functional test H.2.16.5 defined by the IEC 60730 standard. The PC holds the address of the next instruction to be executed.

The test performs the following major tasks:

1. The PC test invokes the functions that are located in the Flash memory at different addresses.
2. These functions reset the error flag.
3. The error flag is tested in many places of the application code.
4. If the error flag is cleared, the PC branches to the correct location.

API FUNCTIONS

The following API functions implement the PC test:

- CLASSB_CPUPCTest()
- CLASSB_CPUPCTestGetResult()
Invariable Memory (Flash/EEPROM) Test

The Invariable Memory (Flash/EEPROM) test implements the periodic modified checksum H.2.19.3.1 defined by the IEC 60730 standard. It detects the single-bit faults in the invariable memory. The invariable memory in a system, such as Flash and EEPROM memory, contains data that is not intended to change during the program execution. The Flash/EEPROM Invariable Memory test computes the periodic checksum using the Cyclic Redundancy Check (CRC-16). The CRC polynomial used to calculate the CRC-16 is shown below.

\[ \text{CRC-16} = 1 1000 0000 0000 0101 \] \[ = 8005 \text{ (hex)} \]

The CRC calculation function returns the final CRC value that can be used to perform the following:

1. If CRC_Flag is set to 0x00 at system start-up, the reference CRC checksum is computed.
2. The reference checksum is stored in the Flash or EEPROM memory and the CRC flag is set to 0xFF.
3. The CRC calculation function can be called periodically if the CRC flag is set to 0xFF.
4. The checksum calculated from step 3 is compared with the reference checksum.
5. If both values match, a status bit can be set by the user application to indicate that the invariable memory has passed the test and no errors were found.

API FUNCTIONS

The following API functions implement the Invariable Memory test:

- `CLASSB_CRCFlashTest`
- `CLASSB_CRCEEPROMTest`

Note: The 16-bit EEPROM test applies only to dsPIC30F devices.

### FIGURE 1: FLOWCHART FOR THE INvariable MEMORY TEST

```
Start

No

CRCFlag == 0

Yes

Calculate the CRC

No

Reference CRC == Calculated CRC

Yes

Pass/No Errors Found

No

Fail/Errors Found

Calculate the Reference CRC Checksum

Yes

Store the Reference CRC Checksum in the Flash/EEPROM Memory

No

Set CRCFlag = 0xFF

End
```
Variable Memory Test

The Variable Memory test implements the Periodic Static Memory test H.2.19.6 defined by the IEC 60730 standard. It detects single bit Faults in variable memory. The variable memory contains data, which is intended to vary during program execution. The RAM Memory test is used to determine if any bit of the RAM memory is stuck at ‘1’ or ‘0’. The March Memory and Checkerboard tests are widely used static memory algorithms for checking the DC Faults.

The following tests can be implemented using the Class B Safety Software Library:

- March Test
  - March C/C- Test
  - March B Test

MARCH TEST

A March test performs a finite set of operations on every memory cell in a memory array. Each operation performs the following tasks:

1. Writes ‘0’ to a memory cell (w0).
2. Writes ‘1’ to a memory cell (w1).
3. Reads the expected value ‘0’ from a memory cell (r0).
4. Reads the expected value ‘1’ from a memory cell (r1).

March Test Notations

Figure 2 illustrates the notations that are used in the March test.

FIGURE 2: MARCH TEST NOTATIONS

\begin{itemize}
  \item $\uparrow$: Arranges the address sequence in ascending order.
  \item $\downarrow$: Arranges the address sequence in descending order.
  \item $\uparrow\downarrow$: Arranges the address sequence in either ascending or descending order.
  \item $r_0$: Indicates a read operation (reads ‘0’ from a memory cell).
  \item $r_1$: Indicates a read operation (reads ‘1’ from a memory cell).
  \item $w_0$: Indicates a write operation (writes ‘0’ to a memory cell).
  \item $w_1$: Indicates a write operation (writes ‘1’ to a memory cell).
\end{itemize}

MARCH C/C- TEST

The March C/C-test is used to detect the following types of Fault in the variable memory:

- Stuck-at Fault
- Addressing Fault
- Transition Fault
- Coupling Fault

The complexity of the March C/C-test is $11n$ and $10n$ respectively, where $n$ indicates the number of bits in the memory. This test can be run as either destructive or non-destructive. If run in non-destructive mode buffer space is required to store the contents of the memory to be tested and restored. If needed, this test can be executed at the system start-up before initializing the memory.

Example 1 shows the pseudocode that demonstrates the implementation of the March C test.

API FUNCTIONS

This API function implements the March C/C-test:

```c
CLASSB_RAMMarchCTest
```

Figure 3 illustrates a March C algorithm.

FIGURE 3: MARCH C ALGORITHM

```
MarchC
{
 \{w0\}; \{r0, w1\}; \{r1, w0\};
 \{r0, w1\}; \{r1, w0\}; \{r0\}
}
```

Note: This step can be skipped for C-.
EXAMPLE 1: PSEUDO CODE FOR MARCH C TEST

/* Ascending: Write 0 */
for(i=0;i<=(n-1);i++)
  x(i)=0;

/* Ascending: Read 0, Write 1 */
for(i=0;i<=(n-1);i++)
{
  if (x(i)==0)
    x(i)=1;
  else
    return fail;
}

/* Ascending: Read 1, Write 0 */
for(i=0;i<=(n-1);i++)
{
  if(x(i)==1)
    x(i)=0;
  else
    return fail;
}

/* Standard March C only */
/* Ascending: Read 0 */
if ( minus != 0)
{
  for(i=(n-1);i>=0;i--)
  {
    if(x(i)==0) {}
    else
      return fail ;
  }
}

/* Descending: Read 0, Write 1 */
for(i=(n-1);i>=0;i--)
{
  if(x(i)==0)
    x(i)=1;
  else
    return fail;
}

/* Descending: Read 1, Write 0 */
for(i=(n-1);i>=0;i--)
{
  if(x(i)==1)
    x(i)=0;
  else
    return fail;
}

/* Ascending: Read 0 */
for(i=(n-1);i>=0;i--)
{
  if(x(i)==0) {}
  else
    return fail;
}
return pass;
MARCH B TEST
The March B is a non-redundant test that can detect the following types of fault:

• Stuck-at
• Linked Idempotent Coupling
• Inversion Coupling

This test is of complexity 17n, where n indicates the number of bits in the memory. This test can be run as either destructive or non-destructive. If run in non-destructive mode buffer space is required to store the contents of the memory to be tested and restored. If needed, this test can be executed at the system start-up before initializing the memory.

Figure 4 illustrates a March B algorithm.

**FIGURE 4: MARCH B ALGORITHM**

```plaintext
MarchB
{
  m(w0); m(r0, w1, r1, w0, r0, w1); m(r1, w0, w1);
  m(r1, w0, w1, w0); m(r0, w1, w0);
}
```

Example 2 shows the pseudocode that demonstrates the implementation of the March B test.

**API FUNCTIONS**
This API function implements the March B test:

```plaintext
CLASSB_RAMMarchBTest
```

EXAMPLE 2: PSEUODOCODE FOR MARCH B TEST

/* Write 0 */
for(i=0;i<=(n-1);i++)
    x(i)=0;

/* Ascending: Read 0, Write 1; Read 1, Write 0; Read 0, Write 1 */
for(i=0;i<=(n-1);i++)
{
    if(x(i)==0)
    {
        x(i)=1;
    }
    else
        return fail;
    if(x(i)==1)
    {
        x(i)=0;
    }
    else
        return fail;
    if(x(i)==0)
    {
        x(i)=1;
    }
    else
        return fail;
/* Ascending: Read 1, Write 0; Write 1 */
for(i=0;i<=(n-1);i++)
{
    if(x(i)==1)
    {
        x(i)=0;
        x(i)=1;
    }
    else
        return fail;
/* Descending: Read 1, Write 0, Write 1, Write 0 */
for(i=(n-1);i>=0;i--)
{
    if(x(i)==1)
    {
        x(i)=0;
        x(i)=1;
        x(i)=0;
    }
    else
        return fail;
/* Descending: Read 0, Write 1, Write 0; */
for(i=(n-1);i>=0;i--)
{
    if(x(i)==0)
    {
        x(i)=1;
        x(i)=0;
    }
    else
        return fail;
}
return pass;
CHECKERBOARD RAM TEST

The Checkerboard RAM test writes the checkerboard patterns to a sequence of adjacent memory locations. This test is performed in units (memory chunks) of 4 bytes. This is a non-destructive memory test.

This test performs the following major tasks:

1. Saves the contents of the memory locations to be tested in the CPU registers.
2. Writes the binary value (length is dependent upon architecture) 101010... to the memory location, 'N', and the inverted binary value, 010101..., to the memory location, 'N+1', and so on, until the whole memory chunk is filled.
3. Reads the contents of all the memory locations in the current chunk and verifies its contents. If the values match, the function continues; otherwise it stops and returns an error.
4. Step 2 and 3 are repeated by writing the inverted pattern to the same locations.
5. Once a memory chunk is completed the test of the next chunk is started until all of the requested memory area is tested.

API FUNCTIONS

This API function implements the Checkerboard RAM test:

CLASSB_RAMCheckerboardTest

Clock Test

According to the IEC 60730 standard, only harmonics and subharmonics of the clock need to be tested. The Clock test implements the independent time slot monitoring H.2.18.10.4 defined by the IEC 60730 standard. It verifies the reliability of the system clock (i.e., the system clock should be neither too fast nor too slow).

The Clock Test function is used to verify the proper operation of the CPU clock.

This test performs the following major tasks:

1. The independent clock source (or a reference clock source) is required for the test. It can be a secondary oscillator or power line. The reference clock should be connected to a timer such as Timer1.
2. During the test the number of CPU clock cycles per the one reference clock period are counted.
3. If the number of clock cycles is outside a specified range, the function returns an error code.

API FUNCTIONS

This API function implements the Clock test:

CLASSB_ClockTest

Addressing of Variable and Invariable Memory and Internal Data Path

For single chip microcontrollers or digital signal controllers, such as PIC MCUs and dsPIC DSCs, the Periodic Static Memory test is used to test the variable memory, and the periodic checksum is used to test the invariable memory. These tests detect any stuck-at Fault in the internal address bus and internal data path.
Addressing Wrong Address
This test is required only for microcontrollers with an external memory device.

External Communication
The IEC 60730 Class B specifications suggest the following measures to ensure reliable communication between components:

TRANSFER REDUNDANCY
The transfer redundancy is a Fault/error control technique that protects against coincidental and/or systematic errors in the input and output information. It is achieved by transferring the data between the transmitter and receiver. The data is transferred at least twice in succession and then compared.

PROTOCOL TEST
The Protocol test is a Fault/error control technique in which the data is transferred to and from the computer components to detect errors in the internal communication protocol.

CRC SINGLE WORD
A CRC polynomial is used to calculate the CRC checksum of the transmitted message. At the transmitting end, this CRC checksum is appended to the message before transmitting it. At the receiving end, the receiver uses the same CRC polynomial to compute the CRC checksum, and compares the computed value with the received value.

Timing
The PIC MCUs and dsPIC DSCs have several dedicated communication interfaces, such as UART, I²C™ and SPI modules. The IEC 60730 Class B specifications suggest that these modules should use time slot monitoring to ensure that the communication occurs at the correct point in time.

Plausibility Check
The plausibility checks on the I/O periphery, analog multiplexer and A/D convertor can be performed as follows:

I/O PERIPHERY
The plausibility check on an I/O pin can be performed by toggling the I/O and checking the state of the pin.

ANALOG MULTIPLEXER
To verify the operation of the analog multiplexer, known voltage values are applied to all channels. These values are read and compared with the applied voltage for verification.

A/D CONVERTER
To test the analog functions of the A/D converter, a known external voltage is applied to the analog inputs. The conversion results are then compared with the applied voltage.
CLASSB_CPURegistersTest

Description
This function implements the CPU Register test. The test writes the values 0x5555 and 0xAAAA into the CPU registers and then reads the values from these registers for verification. The function returns a non-zero if the values do not match. The results are returned into the W0 register. Therefore the contents are the W0 register are not preserved. The content of the CPU register to be tested is saved and then restored upon the completion of the each register test.

Prototype
CLASSBRESULT CLASSB_CPURegistersTest();

Arguments
None

Return Value
CLASSB_TEST_PASS (returned value = 0) – the test finished successfully
CLASSB_TEST_FAIL (returned value != 0) – the test is failed

Source File
classb_registers_.s

TABLE 1: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Flash Memory</td>
<td>1054 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>240 Instruction Cycles</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled for 39 Instruction Cycles</td>
</tr>
</tbody>
</table>
### CLASSB_CPUPCTest

#### Description
This function executes the Program Counter (PC) test, which is a functional test of the PC. The test invokes functions that are located in the Flash memory at different addresses. The pointer of the error flag stored in the RAM is passed into these functions. The functions clear the error flag before returning. If the error flag is not cleared it means that PC was corrupt and the functions were not called. To check the PC error flag another function `CLASSB_CPUPCTestGetResult()` should be used. The addresses of the test functions are based on the device flash memory size. The flash memory size should be specified in `classb_config.h` file using `CLASSB_DEVICE_FLASH_SIZE_KB` parameter.

#### Prototype
```c
void CLASSB_CPUPCTest();
```

#### Arguments
None

#### Return Value
None

#### Source File
classb_pc_.s

#### TABLE 2: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Flash Memory</td>
<td>41 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>32 Instruction Cycles</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled for 32 Instruction Cycles</td>
</tr>
</tbody>
</table>
**CLASSB_CPUPCTestGetResult**

**Description**
This function checks the result of the PC test done by `CLASSB_CPUPCTest()` function. This function should be called in many places of the application code, especially before any critical operations.

**Prototype**
`CLASSBRESULT CLASSB_CPUPCTestGetResult();`

**Arguments**
None

**Return Value**
- `CLASSB_TEST_PASS` (returned value = 0) – the test finished successfully
- `CLASSB_TEST_FAIL` (returned value != 0) – the test is failed

**Source File**
classb.h

**Remarks**
None

**TABLE 3: RESOURCE REQUIREMENTS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Flash Memory</td>
<td>N/A</td>
</tr>
<tr>
<td>Execution Time</td>
<td>N/A</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Enabled</td>
</tr>
</tbody>
</table>
CLASSB_CRCFlashTest

Description
This function implements the Invariable Memory test. It computes the CRC of data starting at `startAddress` for `length` bytes using the `crcSeed` provided. This function returns the final CRC value.

Prototype
```c
uint16_t CLASSB_CRCFlashTest(startAddress, length, crcSeed)
```

Arguments
- `startAddress` - the first address of the tested memory (must be even number)
- `length` - the byte length of the tested memory (must be even number)
- `crcSeed` - initial value of the CRC check sum

Return Value
`crc_Result` Holds the CRC result

Source File
classb_crc.c

Remarks
None

TABLE 4: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Memory</td>
<td>383 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>160000 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Enabled</td>
</tr>
</tbody>
</table>
CLASSB_CRCEEPROMTest

Description
The function calculates the CRC check sum for the EEPROM memory region. The variables `startAddress` and `length` must be even numbers.

Prototype
```c
uint16_t CLASSB_CRCEEPROMTest (startAddress, length, crcSeed)
```

Arguments
- `startAddress` - the first address of the tested EEPROM memory in bytes (must be even number)
- `length` - the byte length of the tested EEPROM memory (must be even number)
- `crcSeed` - initial value of the CRC check sum

Return Value
The function returns the standard 16-bit CRC check sum

Remarks
None

Source File
classb_crc.c

TABLE 5: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Memory</td>
<td>383 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>128000 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Enabled</td>
</tr>
</tbody>
</table>
### CLASSB_RAMMarchCTest

#### Description

This function implements the March C and March C minus tests for the RAM memory. The variables `startAddress` and `length` must be even numbers. This test can be run as either destructive or non-destructive. When the buffer to store the memory content is not specified, it is run in destructive mode and the memory content is not saved. To run in non-destructive mode the buffer address must be defined. The tested RAM region must not cross the storage buffer region. Also this test can be executed at the system start-up before initializing the memory. The `CLASSB_MARCH_C_STARTUP` or `CLASSB_MARCH_C_MINUS_STARTUP` parameters in `classb_config.h` must be set to non-zero value to run the test at the system start-up. The interrupts are disabled during the test.

#### Prototype

```
CLASSBRESULT CLASSB_RAMMarchCTest (startAddress, length, bufferAddress, minus)
```

#### Arguments

- `startAddress` - the first address of the tested memory (must be even number)
- `length` - the byte length of the tested memory (must be even number)
- `bufferAddress` - the first address of the buffer to store the tested memory content (must be even number); if this parameter is NULL then tested memory will be cleared
- `minus` - if the parameter is non-zero, the “minus” algorithm is used.

#### Return Value

- `CLASSB_TEST_PASS` (returned value = 0) - the test finished successfully
- `CLASSB_TEST_FAIL` (returned value != 0) - the test is failed

#### Remarks

None

#### Source File

`classb_marchc_.s`

### TABLE 6: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>1364 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>March C 115,200 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td></td>
<td>March C- 121600 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled during the test</td>
</tr>
</tbody>
</table>
CLASSB_RAMMarchBTest

Description
This function implements the March B test for the RAM memory. The variables `startAddress` and `length` must be even numbers. This test can be run as either destructive or non-destructive. When the `bufferAddress` is not specified, it is run in destructive mode and the memory content is not saved. To run in non-destructive mode, the `bufferAddress` must be defined. The tested RAM region must not cross the storage buffer region. Also, this test can be executed at the system start-up before initializing the memory. The `CLASSB_MARCH_B_STARTUP` parameter in `classb_config.h` must be set to non-zero value to run the test at the system start-up. The interrupts are disabled during the test.

Prototype
CLASSBRESULT CLASSB_RAMMarchBTest(startAddress, length, bufferAddress);

Arguments
• `startAddress` - the first address of the tested memory (must be even number)
• `length` - the byte length of the tested memory (must be even number)
• `bufferAddress` - the first address of the buffer to store the tested memory content (must be even number); if this parameter is NULL then tested memory will be cleared

Return Value
CLASSB_TEST_PASS (returned value = 0) - the test finished successfully
CLASSB_TEST_FAIL (returned value != 0) - the test is failed

Remarks
None

Source File
classb_marchb_.s

TABLE 7: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Memory</td>
<td>2003 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>192000 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled during the test</td>
</tr>
</tbody>
</table>
CLASSB_RAMCheckerboardTest

Description
This function implements the Checkerboard test on the RAM memory. The test is performed on length bytes starting at startAddress. The variable length must be divisible by 4.

Prototype
CLASSBRESULT CLASSB_RAMCheckerboardTest(startAddress, length)

Arguments
• startAddress - the first address of the tested memory (must be even number)
• length - the byte length of the tested memory (must be divisible by 4 bytes).

Return Value
CLASSB_TEST_PASS (returned value = 0) - the test finished successfully
CLASSB_TEST_FAIL (returned value != 0) - the test is failed

Remarks
None.

Source File
classb_checkerboard_.s

TABLE 8: RESOURCE REQUIREMENTS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Memory</td>
<td>190 Bytes</td>
</tr>
<tr>
<td>Stack</td>
<td>8000 Instruction Cycles per Kbyte</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled for 24 Instruction Cycles</td>
</tr>
</tbody>
</table>
**CLASSB_ClockTest**

**Description**

This function implements the Clock test. It is used to verify that the CPU clock source is operating within the acceptable frequency tolerance. The reference clock (e.g., external crystal or line frequency) needs to be connected to a timer. The timer must be initialized by the application code to count the reference clock pulses. The period register must be set to the maximum value. The address of the timer counter register must be specified in `classb_config.h` using the compile time option `CLASSB_CLOCK_TEST_TIMER_ADDRESS`. The compile option `CLASSB_CLOCK_TEST_TIME_MS` in the `classb_config.h` file defines the test time – it can be about 20 ms if the clock is between 1-32 MHz and the reference clock is 50 Hz-33 kHz. Interrupts are disabled during the test.

**Prototype**

```c
CLASSB_ClockTest (uint32_t clockFrequency, uint32_t referenceFrequency, uint16_t tolerance)
```

**Arguments**

- `clockFrequency` - frequency of the clock source
- `referenceFrequency` - frequency of the reference clock (such as power line or secondary oscillator)
- `tolerance` - maximum valid frequency tolerance, can be from 1(0.1%) to 100(10%)

**Return Value**

- `CLASSB_TEST_PASS` (returned value = 0) - the test finished successfully
- `CLASSB_TEST_FAIL` (returned value != 0) - the test is failed

**Remarks**

None.

**Source File**

- `classb_clock_.s`
- `classb_clock.c`

**TABLE 9: RESOURCE REQUIREMENTS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash Memory</td>
<td>633 Bytes</td>
</tr>
<tr>
<td>Execution Time</td>
<td>160000-640000 Instruction Cycles</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Disabled during the test</td>
</tr>
</tbody>
</table>
SUMMARY

This application note describes how to implement various diagnostic measures proposed by the IEC 60730 standard. These measures ensure the safe operation of controlled equipment that falls under the Class B category. In addition, this application note also describes the different APIs that are available in the Class B Safety Software Library. These APIs can be directly integrated with the end user’s application to test and verify the critical functionalities of a controller and are intended to maximize the application reliability through Fault detection. When implemented on a dsPIC DSC or PIC MCU, these APIs help meet the IEC 60730 standard's requirements.

Microchip has developed the Class B Safety Software Library to assist you in implementing the safety software routines. Contact your Microchip sales or application engineer if you would like further support.

REFERENCES

• IEC 60730 Standard, “Automatic Electrical Controls for Household and Similar Use”, IEC 60730-1 ed.5, 2013
• Wu, C. “Memory Testing”
• Wu, C. “RAM Fault Models and Memory Testing”
APPENDIX A: SOURCE CODE

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the “Company”) is intended and supplied to you, the Company’s customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

All of the software covered in this application note is available as a single zip file. This archive can be downloaded from the Microchip corporate web site at:

www.microchip.com/classb
### APPENDIX B: IEC 60730-1 TABLE H.1 (H.11.12.7 OF EDITION 3)

The following table is reproduced with the permission of the International Electrotechnical Commission (IEC). IEC 60730-1 ed.5 “Copyright © 2013 IEC, Geneva, Switzerland. www.iec.ch”.

#### TABLE B-1: Table H.1 (H.11.12.7 of edition 3) – Acceptable measures to address fault/errors

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td>rq</td>
<td>Functional test, or</td>
<td>H.2.16.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>periodic self-test using either:</td>
<td>H.2.16.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- static memory test, or</td>
<td>H.2.19.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- word protection with single bit redundancy</td>
<td>H.2.19.8.2</td>
</tr>
<tr>
<td>DC fault</td>
<td></td>
<td>rq</td>
<td>Comparison of redundant CPUs by either:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- reciprocal comparison</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>internal error detection, or</td>
<td>H.2.18.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>redundant memory with comparison, or</td>
<td>H.2.19.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>periodic self-tests using either</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- walkpat memory test</td>
<td>H.2.19.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- Abraham test</td>
<td>H.2.19.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- transparent GALPAT test; or</td>
<td>H.2.19.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>word protection with multi-bit redundancy, or</td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>static memory test and word protection with single bit redundancy</td>
<td>H.2.19.6</td>
</tr>
</tbody>
</table>

| 1.2       | Instruction decoding and execution | rq | Comparison of redundant CPUs by either: | H.2.18.15 |
|           | Wrong decoding and execution       | rq | - reciprocal comparison                | H.2.18.15 |
|           |                                     | rq | - independent hardware comparator, or   | H.2.18.3 |
|           |                                     | rq | internal error detection, or            | H.2.18.9 |
|           |                                     | rq | periodic self-test using equivalence class test | H.2.18.5 |

| 1.3       | Programme counter                  | rq | Functional test, or                     | H.2.16.5    |
|           |                                       | rq | periodic self-test, or                   | H.2.16.6    |
|           |                                       | rq | independent time-slot monitoring of the  | H.2.16.10.4 |
|           |                                       | rq | program sequence, or                     |             |
|           |                                       | rq | logical monitoring of the programme sequence, or |             |
|           |                                       | rq | Periodic self-test and monitoring using either: |             |
|           |                                       | rq | - independent time-slot and logical       | H.2.16.7    |
|           |                                       | rq | monitoring                               | H.2.18.10.3 |
|           |                                       | rq | - internal error detection, or           | H.2.18.9    |
|           |                                       | rq | comparison of redundant functional channels by |             |
|           |                                       | rq | either:                                  |             |
|           |                                       | rq | - reciprocal comparison                  | H.2.18.15   |
|           |                                       | rq | - independent hardware comparator        | H.2.18.3    |

CPU: Central programmation unit

rq: Coverage of the fault is required for the indicated software class.

a Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.
b For fault/error assessment, some components are divided into their subfunctions.
c For each subfunction in the table, the software class C measure will cover the software class B fault/error.
d It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.
e Where more than one measure is given for a subfunction, these are alternatives.
f To be divided as necessary by the manufacturer into subfunctions.
### TABLE B-1: Table H.1 (H.11.12.7 of edition 3) – Acceptable measures to address fault/errors

<table>
<thead>
<tr>
<th>Componentb</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measuresc d e</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4</td>
<td>Addressing</td>
<td>rq</td>
<td>Comparison of redundant CPUs by either:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator; or</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- internal error detection; or</td>
<td>H.2.18.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>periodic self-test using a testing pattern of the address lines; or</td>
<td>H.2.16.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full bus redundancy or multi-bit bus parity</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td>1.5</td>
<td>Data paths instruction decoding</td>
<td>rq</td>
<td>Comparison of redundant CPUs by either:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DC fault and execution</td>
<td></td>
<td>- reciprocal comparison, or</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- internal error detection, or</td>
<td>H.2.18.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>periodic self-test using a testing pattern, or</td>
<td>H.2.16.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data redundancy, or multi-bit bus parity</td>
<td>H.2.18.2.1</td>
</tr>
<tr>
<td>2.</td>
<td>Interrupt handling and execution</td>
<td>rq</td>
<td>Functional test; or time-slot monitoring</td>
<td>H.2.16.5</td>
</tr>
<tr>
<td></td>
<td>No interrupt or too frequent</td>
<td>rq</td>
<td>Comparison of redundant functional channels by either</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
<td></td>
<td>- reciprocal comparison,</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td>No interrupt or too frequent</td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.9</td>
</tr>
<tr>
<td></td>
<td>interrupt related to different</td>
<td></td>
<td>- independent time-slot and logical monitoring</td>
<td>H.2.18.10.3</td>
</tr>
<tr>
<td></td>
<td>sources</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Clock</td>
<td>rq</td>
<td>Frequency monitoring, or time-slot monitoring</td>
<td>H.2.18.10.1</td>
</tr>
<tr>
<td></td>
<td>Wrong frequency</td>
<td>rq</td>
<td>Frequency monitoring, or time-slot monitoring, or comparison of redundant functional channels by either:</td>
<td>H.2.18.10.4</td>
</tr>
<tr>
<td></td>
<td>(for quartz synchronized clock:</td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td>harmonics/subharmonics only)</td>
<td></td>
<td>- independent hardware comparator</td>
<td>H.2.18.3</td>
</tr>
</tbody>
</table>

CPU: Central programation unit

rq: Coverage of the fault is required for the indicated software class.

a Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.

b For fault/error assessment, some components are divided into their subfunctions.

c For each subfunction in the table, the software class C measure will cover the software class B fault/error.

d It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.

e Where more than one measure is given for a subfunction, these are alternatives.

f To be divided as necessary by the manufacturer into subfunctions.
### TABLE B-1: Table H.1 (H.11.12.7 of edition 3) – Acceptable measures to address fault/errors

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. Memory</td>
<td></td>
<td></td>
<td></td>
<td>H.2.19.3.1</td>
</tr>
<tr>
<td>4.1</td>
<td></td>
<td></td>
<td></td>
<td>H.2.19.3.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H.2.19.8.2</td>
</tr>
<tr>
<td>Invariable memory</td>
<td>All single bit faults</td>
<td>rq</td>
<td>Periodic modified checksum; or multiple checksum, or word protection with single bit redundancy</td>
<td>H.2.19.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>Comparison of redundant CPUs by either: - reciprocal comparison, or independent hardware comparator, or redundant memory with comparison, or periodic cyclic redundancy check, either: - single word, or double word, or word protection with multi-bit redundancy</td>
<td>H.2.19.4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td>4.2</td>
<td>DC fault</td>
<td>rq</td>
<td>Periodic static memory test, or word protection with single bit redundancy</td>
<td>H.2.19.6</td>
</tr>
<tr>
<td>Variable memory</td>
<td>DC fault and dynamic cross links</td>
<td>rq</td>
<td>Comparison of redundant CPUs by either: - reciprocal comparison, or independent hardware comparator, or redundant memory with comparison, or periodic self-tests using either: - walkpat memory test, - Abraham test, - transparent GALPAT test, or word protection with multi-bit redundancy</td>
<td>H.2.19.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td>4.3</td>
<td>Stuck at DC fault</td>
<td>rq</td>
<td>Word protection with single bit redundancy including the address, or comparison of redundant CPUs by either: - reciprocal comparison, or independent hardware comparator, or full bus redundancy</td>
<td>H.2.19.18.2</td>
</tr>
<tr>
<td>Addressing (relevant to variable memory and invariable memory)</td>
<td></td>
<td>rq</td>
<td>Testing pattern, or periodic cyclic redundancy check, either: - single word, or double word, or word protection with multi-bit redundancy including the address</td>
<td>H.2.19.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.18.1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.18.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td></td>
<td>H.2.19.8.1</td>
</tr>
</tbody>
</table>

**CPU:** Central programation unit

**rq:** Coverage of the fault is required for the indicated software class.

**a** Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.

**b** For fault/error assessment, some components are divided into their subfunctions.

**c** For each subfunction in the table, the software class C measure will cover the software class B fault/error.

**d** It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.

**e** Where more than one measure is given for a subfunction, these are alternatives.

**f** To be divided as necessary by the manufacturer into subfunctions.
### 5. Internal data path

#### 5.1 Data

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stuck at DC fault</td>
<td>rq</td>
<td>Word protection with single bit redundancy</td>
<td>H.2.19.8.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Comparison of redundant CPUs by either:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>word protection with multi-bit redundancy</td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>including the address, or data redundancy, or</td>
<td>H.2.18.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>testing pattern, or</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>protocol test</td>
<td>H.2.18.14</td>
</tr>
</tbody>
</table>

#### 5.2 Addressing

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wrong address</td>
<td>rq</td>
<td>Word protection with single bit redundancy</td>
<td>H.2.19.8.2</td>
</tr>
<tr>
<td></td>
<td>Wrong address and multiple addressing</td>
<td>rq</td>
<td>including the address</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Comparison of redundant CPUs by:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>word protection with multi-bit redundancy,</td>
<td>H.2.18.1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>including the address, or full bus redundancy; or</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>testing pattern including the address</td>
<td></td>
</tr>
</tbody>
</table>

### 6. External communication

#### 6.1 Data

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hamming distance 3</td>
<td>rq</td>
<td>Word protection with multi-bit redundancy, or CRC - single word, or transfer redundancy, or protocol test</td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>H.2.19.4.1</td>
<td>H.2.18.2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>H.2.18.14</td>
<td></td>
</tr>
</tbody>
</table>

#### 6.2 Addressing

<table>
<thead>
<tr>
<th>Component</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wrong address</td>
<td>rq</td>
<td>CRC - double word, or data redundancy or comparison of redundant functional channels by either:</td>
<td>H.2.19.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td>Wrong and multiple addressing</td>
<td>rq</td>
<td>Word protection with multi-bit redundancy, including the address, or CRC - single word, including the addresses, or transfer redundancy or protocol test</td>
<td>H.2.19.8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CRC - double word, including the address, or full bus redundancy of data and address, or comparison of redundant communication channels by either:</td>
<td>H.2.19.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator</td>
<td>H.2.18.1.1</td>
</tr>
</tbody>
</table>

### Definitions

- **CPU:** Central programmation unit
- **rq:** Coverage of the fault is required for the indicated software class.
- **a** Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.
- **b** For fault/error assessment, some components are divided into their subfunctions.
- **c** For each subfunction in the table, the software class C measure will cover the software class B fault/error.
- **d** It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.
- **e** Where more than one measure is given for a subfunction, these are alternatives.
- **f** To be divided as necessary by the manufacturer into subfunctions.
### TABLE B-1: Table H.1 (H.11.12.7 of edition 3) – Acceptable measures to address fault/errors\(^a\)

<table>
<thead>
<tr>
<th>Component(^b)</th>
<th>Fault/error</th>
<th>Software Class</th>
<th>Example of acceptable measures(^c)(^d)(^e)</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3 Timing</td>
<td>Wrong point in time</td>
<td>rq</td>
<td>Time-slot monitoring, or scheduled transmission</td>
<td>H.2.18.10.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time-slot and logical monitoring, or comparison of redundant communication channels by either:</td>
<td>H.2.18.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>reciprocal comparison</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>independent hardware comparator</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td>Wrong sequence</td>
<td>rq</td>
<td>Logical monitoring, or time-slot monitoring, or scheduled transmission</td>
<td>H.2.18.10.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(same options as for wrong point in time)</td>
<td>H.2.18.18</td>
</tr>
<tr>
<td>7. Input/output periphery</td>
<td>Fault conditions specified in Clause H.27</td>
<td>rq</td>
<td>Plausibility check</td>
<td>H.2.18.13</td>
</tr>
<tr>
<td>7.1 Digital I/O</td>
<td></td>
<td>rq</td>
<td>Comparison of redundant CPUs by either:</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>input comparison, or multiple parallel outputs; or</td>
<td>H.2.18.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output verification, or testing pattern, or</td>
<td>H.2.18.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>code safety</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td>7.2 Analog I/O</td>
<td>Fault conditions specified in Clause H.27</td>
<td>rq</td>
<td>Plausibility check</td>
<td>H.2.18.13</td>
</tr>
<tr>
<td>7.2.1 A/D- and D/A- convertor</td>
<td></td>
<td>rq</td>
<td>Comparison of redundant CPUs by either:</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- reciprocal comparison</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input comparison, or multiple parallel outputs, or</td>
<td>H.2.18.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>output verification, or testing pattern</td>
<td>H.2.18.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>code safety</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td>7.2.2 Analog multiplexer</td>
<td>Wrong addressing</td>
<td>rq</td>
<td>Plausibility check</td>
<td>H.2.18.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Comparison of redundant CPUs by either:</td>
<td>H.2.18.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rq</td>
<td>- reciprocal comparison</td>
<td>H.2.18.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- independent hardware comparator, or</td>
<td>H.2.18.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>input comparison or testing pattern</td>
<td>H.2.18.22</td>
</tr>
<tr>
<td>8. Monitoring devices and comparators</td>
<td>Any output outside the static and dynamic functional specification</td>
<td>rq</td>
<td>Tested monitoring, or redundant monitoring and comparison, or error recognizing means</td>
<td>H.2.18.21</td>
</tr>
</tbody>
</table>

---

**CPU:** Central programming unit

**rq:** Coverage of the fault is required for the indicated software class.

**a**: Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.

**b**: For fault/error assessment, some components are divided into their subfunctions.

**c**: For each subfunction in the table, the software class C measure will cover the software class B fault/error.

**d**: It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.

**e**: Where more than one measure is given for a subfunction, these are alternatives.

**f**: To be divided as necessary by the manufacturer into subfunctions.
### Table B-1: Table H.1 (H.11.12.7 of edition 3) – Acceptable measures to address fault/errors\(^a\)

<table>
<thead>
<tr>
<th>Component(^b)</th>
<th>Fault/error</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. Custom chips(^f) for example, ASIC, GAL, Gate array</td>
<td>Any output outside the static and dynamic functional specification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Class</th>
<th>Example of acceptable measures(^c) (^d) (^e)</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>rq</td>
<td>rq</td>
<td>Periodic self-test</td>
</tr>
<tr>
<td>rq</td>
<td></td>
<td>Periodic self-test and monitoring, or dual channel (diverse) with comparison, or error recognizing means</td>
</tr>
<tr>
<td>rq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Definitions**
- CPU: Central programation unit
- rq: Coverage of the fault is required for the indicated software class.
- a: Table H.1 is applied according to the requirements of H.11.12 to H.11.12.2.12 inclusive.
- b: For fault/error assessment, some components are divided into their subfunctions.
- c: For each subfunction in the table, the software class C measure will cover the software class B fault/error.
- d: It is recognized that some of the acceptable measures provide a higher level of assurance than is required by this standard.
- e: Where more than one measure is given for a subfunction, these are alternatives.
- f: To be divided as necessary by the manufacturer into subfunctions.
APPENDIX C: REVISION HISTORY

Revision A (August 2014)
This is the initial release of this application note.
Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Microchip makes no representations or warranties of any kind, whether express or implied, written or oral, statutory or otherwise, related to the information, including but not limited to its condition, quality, performance, merchantability or fitness for purpose. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademark

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, flexPWR, JukeBlox, KEELOQ, KEELOQ logo, Kleer, LANCheck, MediaLB, MOST, MOST logo, MPLAB, OptoLyzer, PIC, PICSTART, PIC32™ logo, RightTouch, SpyNIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
The Embedded Control Solutions Company and mTouch are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, ECAN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, Kleer, KleerNet logo, KleerNet, MWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, RightTouch logo, REAL ICE, SQI, Serial Quad I/O, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestiC is a registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2014, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.


QUALITY MANAGEMENT SYSTEM CERTIFIED BY DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company’s quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KielLoc® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001:2000 certified.