NEC

Application Note

IEC60730 Class B

Support for certification

78K0 Series

Document No. U18617EE1V0AN00
Date published April 2007

Printed in Germany

Legal Notes

- **The information contained in this document is being issued in advance of the production cycle for the product. The parameters for the product may change before final production or NEC Electronics Corporation, at its own discretion, may withdraw the product prior to its production.**

- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: "Standard", "Special", and "Specific". The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics products before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Notes for CMOS Devices

1. **VOLTAGE APPLICATION WAVEFORM AT INPUT PIN**
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between VIL (MAX) and VIH (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between VIL (MAX) and VIH (MIN).

2. **HANDLING OF UNUSED INPUT PINS**
   Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to VDD or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

3. **PRECAUTION AGAINST ESD**
   A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

4. **STATUS BEFORE INITIALIZATION**
   Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

5. **POWER ON/OFF SEQUENCE**
   In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current. The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

6. **INPUT OF SIGNAL DURING POWER OFF STATE**
   Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact theNEC office in your country to obtain a list of authorized representatives anddistributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and otherlegal issues may also vary from country to country.

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668, Japan
Tel: 044 4355111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554,
U.S.A.
Tel: 408 5886000
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211 65030
http://www.eu.necel.com/

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908 691133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01 30675800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091 5042787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 6387200

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02 667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven,
The Netherlands
Tel: 040 2654010

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27
ZhiChunLu Haidian District,
Beijing 100083, P.R.China
Tel: 010 82351155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China
Tower,
200 Yincheng Road Central,
Pudong New Area,
Shanghai 200120, P.R. China
Tel: 021 58885400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
12/F., Cityplaza 4,
12 Taikoo Wan Road, Hong Kong
Tel: 2886 9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R.O.C.
Tel: 02 27192377

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253 8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku, Seoul,
135-080, Korea Tel: 02-558-3737
http://www.kr.necel.com/

Table of Contents

Chapter 1 Overview

1.1 Introduction

Home appliances will have to comply with standard IEC60730 from October 2007 onwards. This requires the inclusion of features that will avoid failure or at least ensure any failure in the appliance does not present a safety hazard to the user. Manufacturers of home appliances must therefore consider how their products will be designed to comply with these requirements.

1.2 IEC60730

IEC60730-1 applies to automatic electrical controls in association with equipment for household and similar use. This includes controls for heating, air-conditioning and similar applications using electricity, gas, oil, solid fuel, solar energy etc. including combinations of it.

Controls shall be designed and constructed that in normal use they function so as not to cause injury to persons or damage to surrounding property, even in the event of such carelessness as may occur in normal use.

More specifically the standard applies to automatic electrical controls, mechanically or electrically operated, responsive to or controlling such characteristics as temperature, pressure, passage of time, humidity, light, electrostatic effects, flow, or liquid level, current, voltage, acceleration, or a combinations of these.

Chapter 2 Electronic Control Requirements

2.1 IEC60730 Annex H

The standard applies to electronic controls and controls using software and therefore as is typically the case in modern appliances, the use of a microcontroller(s). Designers must therefore consider at the outset of their development what features the microcontroller will require in terms of hardware & software in order to comply with the requirements of the standard.

Among others, Annex H of IEC60730 explains the details of test and diagnostic methods for the Microcontrollers.

2.2 Software Classification

The software-related controls are classified by the standard as A, B or C.

Class A - Control functions which are not intended to be relied upon for the safety of the equipment

Class B - Control functions intended to prevent unsafe operation of the controlled equipment. E.g. Thermal cut outs, door locks for laundry equipment.

Class C – Control functions which are intended to prevent special hazards (e.g. explosion of the controlled equipment such as burner controls).

Chapter 3 Class B Requirements

3.1 Software Category B

Category B will apply to the majority of home appliances e.g. clothes washers, dryers, dishwashers, refrigerators, cookers. Category C would be concerned with, for example gas fired boilers.

The specification requires that controls with functions related to software class B or C shall use measures to avoid and control software related faults or errors in safety-related data and safety-related segments of the software. This essentially means the software and/or hardware must employ diagnostic methods to detect faults internally and externally to the microcontroller.

Annex H table H.11.12.7 defines the elements of the microcontroller that must be monitored and tested, the type of faults to be detected and recommends tests to be performed.

3.2 Elements to be tested

NEC has developed a self test library (STL), focusing on Class B requierments for 78K0 devices which covers many IEC60730 requierements listed in the standard.

For Class B controllers, the elements of the controller that must be monitored and tested are defined in Annex H table H.11.12.7 a summary of which is shown below.

CPU	
Register test Programm counter	stuck at stuck at
Interrupt handling and execution	
Interrupt test	no or too frequent inerrupts
System clock	
Clock test	wrong frequency (harmonics)
Memory	
Invariable memory	single bit faults
Variable memory	dc fault
Addressing	covered by memory test
Data path	
Internal data path	only with external memory
Addressing	only with external memory
External communication	
Test	Hamming distance 3
Timing	wrong point in time/sequence

For a detailed description on the individual modules of the self test library pls refer to the API description in chapter 5

Chapter 4 Self Test Library

4.1 STL Approach

In consultance with appliance manufactures and standard authorities, NEC Electronics has considered ways in which its microcontrollers may be used to meet parts of the requierments of Class B control functions.

As a result of this NEC has developed a Self Test Library to support its customers in the Class B certification process of IEC60730 Annex H.

The Libraray provides Application Programmers Interface calls which can be used in customers application wherever requiered, e.g. at system startup or periodically, as the application requires.

The system hardware must provide (at least) two independent clock sources, e.g. crystal/ceramic oscillator and line frequency.

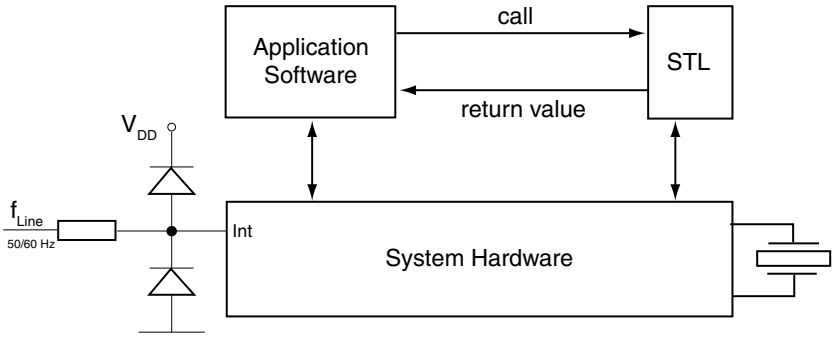

Figure 4-1 STL Approach

Chapter 5 API Description

5.1 Notes

The IEC60730-1 standard describes the requirements of “Automatic Electrical Controls for Household and similar use”. It is an international standard for the certification of electronic control units for white goods and household equipment.

Examples are firing systems, washing machines and ovens, which have potentially dangerous operating or failure modes.

This chapter describes the Application Programmers Interface (API) to the SelfTest Library (STL), which provides the self test functions as required by the IEC60730-1 standard.

This chapter is a collection of implementation ideas, which may help to get a system certified.

The individual STL Funktions are implemented in an Example program.

They can be used with or without modifications or can serve as a discussion basis for our customers.

5.2 API Introduction

Considerations on functional integrity

Software alone cannot detect all possible failures of a device – not even close to all. Semiconductor manufacturers implement a lot of costly hardware overhead on a chip, so that it can be quickly and comprehensively tested after manufacturing. These test structures can only be accessed in specific test modes on a chip tester. They are not available to the CPU in normal operating mode and therefore they cannot be used for a system check. Depending on the device and on the technology, the test structures could account for up to one third of the chip size. The total quality level of a few ppm (i.e. a few failures in a million devices) can only be achieved through a combination of different tests (and applied statistics).

Besides functional testing, i.e. what the STL does, there is one surprisingly effective test, which may be difficult to implement in an application, but perhaps worth considering, the I_{DDQ} (quiescent current) test. Its principle is the measurement of the static current consumption, i.e. while the device is not working. The I_{DDQ} test is based on the fact that a dominant failure mode is the increased leakage current of one transistor in a complementary buffer. If the other transistor is switched on, the total current is increased compared to a working buffer, which ideally draws no static current at all. I_{DDQ} test hardware would stop the device clock, wait until all internal capacitors are charged or discharged and then measure the current consumption. If it is significantly above the quiescent current, something is wrong. Otherwise the clock is toggled, so that as many internal nodes as possible change their level, and the current is measured again. After a suitable number of measurements, a fault coverage of about 70% can be

achieved. It should be noted that the practical implementation of such an IDDQ test is probably challenging.

While the STL functions provide some additional safety by themselves, for optimum fault coverage it is required to implement an external supervisory circuit (e.g. a watchdog timer). That is because some failures will impact the instruction flow sequence and therefore execution may not return to the calling program, i.e. the program hangs. If the watchdog timer triggers, the system should be automatically switched into a fail safe mode without using the CPU.

5.3 STL Overview

The Self Test Library (STL) consists of several independent function modules, which have to be executed cyclically as required by the application. Each function contributes specific system tests and returns the result or an error status. For the best possible test coverage, the user must call each function in defined time intervals, verify its return value and take appropriate action if an error was detected.

5.3.1 Interrupts

For various reasons, interrupts may interfere with the operation of the self test library. The STL needs to write test pattern into the registers and the memory which is to be tested. As interrupts occur asynchronously to the program flow, the interrupt handler would not find the expected contents in these registers. Another problem is that served interrupts will automatically modify system registers, which may currently be tested. The previous register contents may not be restored (e.g. AX, BC, DE, HL on the various Register Banks) so that the self test software would consider the register corrupted and signal a false error.

The STL could simply unconditionally disable interrupts in these cases, but disabling interrupts for a long time may impair system performance and sometimes even the stability.

Whether interrupts need to be disabled or not during the various STL funktions depends highly on the final application software and has to be decided by the user of the STL library.

Pls refer to the interrupt related explanations at the beginning of each Self Test funktion.

Interrupts are not disabled in the existing STL. Proposals are made in the source code, but they are commented out and need to be activated if requiered.

5.3.2 Hardware Reset

A hardware reset may occur at any time and requires special consideration. It will reset the controller and cause program execution to begin at the reset vector. It will not return to the location, at which code was executed, when the reset occurred. It will also not rely on the content of any global register. In that sense, a hardware reset is safe for the integrity of the STL. However, it may not be safe for the system integrity, because the memory may have been under test by the STL when the reset occurred and memory contents may therefore be corrupted. That needs to be considered when RAM content shall be preserved during system restart.

5.3.3 Redundant Data Storage

The self test library provides functions to store and retrieve data with additional redundancy, so that single or multiple bit errors may be detected or even corrected. Reading and writing redundant data requires some processing and memory space overhead, so that it will usually be restricted to the most important safety critical data in the system. Two different algorithms have been implemented, which both require twice as much storage space as the native data (byte or word) that they store.

The simplest, smallest and quickest of these algorithms just stores the original data along with its complement. A read access fetches the original and the complement data, inverts one of them and compares them to each other. Data cannot be corrected with such an algorithm, but errors can be found rather reliably. This strategy is similar to storing a parity bit, but it is superior, because only one of 2^n bit combinations is correct, while single bit parity would not detect 2^n-1 from 2^n combinations, i.e. half of all possible errors would remain undetected. In other words, if a program performs erroneous write operations or a memory bit cells fails, this error would only remain undetected, if the complement was written to the complement's memory location or if the same bit in the complement would fail to the opposite direction. Both cases are rather unlikely.

The second and more demanding algorithm adds an error correction code (ECC) to the data. For performance reasons, an (8,4) Hamming code has been selected. That means that eight bits are used to store four bits of data, hence twice the memory space of the data type is required. The Hamming distance is four, which allows 1 bit errors to be corrected and 2 bit and 3 bit errors to be detected. The ECC write functions split-up the data into two or four 4 bit nibbles, use a lookup table to generate two or four eight 8 bit data values with redundancy and store these values at the target address. The ECC read functions retrieve the encoded data, repair any possible 1 bit error and return the corrected data.

It is important to understand that 3 bit errors cannot be distinguished from 1 bit errors. As a consequence, they are repaired and possibly a wrong value is returned. That is usually acceptable, since even a 1 bit error is an unlikely event. A 2 bit error should be extremely unlikely and the probability of a 3 bit error should be sufficiently close to zero.

One might consider redundancy superior to ECC if no error correction is desired, because it can detect errors of up to n out of 2^n (original and complement) bits, while the implemented ECC can only detect 3 bit errors out of 8 bits. That is a misconception however, as in both cases an entity of n bits is stored in a memory location of 2^n bits. In both cases the number of correct combinations is 2^n, while the total number of combinations is 2^{2n}.

Encoding and decoding are performed via lookup tables. That keeps the code small and the execution times short and predictable. A table with 16 entries is used for encoding a four-bit value, while a 256-entry table is used for decoding it.

The following table is used for encoding a nibble into a byte:

```
0x42,0x4d,0x71,0x7e,0x17,0x18,0x24,0x2b,
0xd4,0xdb,0xe7,0xe8,0x81,0x8e,0xb2,0xbd
```

This table is stored in an array of 16 characters. As an example, the character 0x7b would be stored as 0x2b, 0xe8. The above table is selected in such a way, that the special values 0x00 and 0xff do not appear. It seems that these two values

might be easily generated by mistake and with the implemented table they represent invalid data.

The following lookup table is used for decoding an ECC byte to a 4 bit nibble:

```
0x80,0x4c,0x40,0x80,0x46,0x81,0x80,0x44,0x45,0x81,0x80,0x47,0x81,0x41,0x4d,0x81,
0x45,0x82,0x80,0x44,0x84,0x44,0x44,0x04,0x05,0x45,0x45,0x84,0x45,0x81,0x83,0x44,
0x46,0x82,0x80,0x47,0x06,0x46,0x46,0x84,0x85,0x47,0x47,0x07,0x46,0x81,0x83,0x47,
0x82,0x42,0x4e,0x82,0x46,0x82,0x83,0x44,0x45,0x82,0x83,0x47,0x83,0x4f,0x43,0x83,
0x40,0x80,0x00,0x40,0x80,0x41,0x40,0x80,0x80,0x41,0x40,0x80,0x41,0x01,0x80,0x41,
0x80,0x42,0x40,0x80,0x48,0x81,0x80,0x44,0x45,0x81,0x80,0x49,0x81,0x41,0x43,0x81,
0x80,0x42,0x40,0x80,0x46,0x81,0x80,0x4a,0x4b,0x81,0x80,0x47,0x81,0x41,0x43,0x81,
0x42,0x02,0x80,0x42,0x82,0x42,0x43,0x82,0x82,0x42,0x43,0x82,0x43,0x81,0x03,0x43,
0x4c,0x0c,0x80,0x4c,0x86,0x4c,0x4d,0x84,0x85,0x4c,0x4d,0x87,0x4d,0x81,0x0d,0x4d,
0x85,0x4c,0x4e,0x84,0x48,0x84,0x84,0x44,0x45,0x85,0x85,0x49,0x85,0x4f,0x4d,0x84,
0x86,0x4c,0x4e,0x87,0x46,0x86,0x86,0x4a,0x4b,0x87,0x87,0x47,0x86,0x4f,0x4d,0x87,
0x4e,0x82,0x0e,0x4e,0x86,0x4f,0x4e,0x84,0x85,0x4f,0x4e,0x87,0x4f,0x0f,0x83,0x4f,
0x80,0x4c,0x40,0x80,0x48,0x81,0x80,0x4a,0x4b,0x81,0x80,0x49,0x81,0x41,0x4d,0x81,
0x48,0x82,0x80,0x49,0x08,0x48,0x48,0x84,0x85,0x49,0x49,0x09,0x48,0x81,0x83,0x49,
0x4b,0x82,0x80,0x4a,0x86,0x4a,0x4a,0x0a,0x0b,0x4b,0x4b,0x87,0x4b,0x81,0x83,0x4a,
0x82,0x42,0x4e,0x82,0x48,0x82,0x83,0x4a,0x4b,0x82,0x83,0x49,0x83,0x4f,0x43,0x83
```

The low order nibble is the decoded 4 bit value, but it is only valid if bit 7 is clear. Bit 7 is set if a 2 bit error has been encountered, which cannot be repaired. A 1 bit error can be fixed and it is indicated by bit 6 being set.

Examples: 0x2b is decoded as 0x07 and 0xe8 means 0x0b, so that they generate the original character 0x7b from the above example without an error being flagged. 0xbc is corrected to 0x0f, while 0xbb has a 2 bit error and cannot be fixed.

5.3.4 Global Definitions

Definitions for error detection and error correction functions:

```
typedef struct {unsigned char data[2] ;} ECCBYTE ;
typedef struct {unsigned int data[2] ;} ECCWORD ;
typedef struct {unsigned char data[2] ;} COMPBYTE ;
typedef struct {unsigned int data[2] ;} COMPWORD ;
```

The BYTE and WORD structures are implemented for 78K0 architecture.

```
unsigned int comp_error(void *addr, unsigned char size, unsigned int data) ;
```

comp_error is a user defined function, which is executed when a complement error was detected by a comp_xread function. addr points to the corrupted data, size is its size in number of bytes (1 or 2) and data is the possibly corrupted data returned from the comp_xread function. comp_errorhandler is called in a fatal situation, and it would probably move the system into a fail-safe mode or do a re-initialization

```
unsigned int ecc_error(void *addr, unsigned char size, unsigned char data) ;
```

ecc_error is a user defined function, which is executed when an uncorrectable ECC error was detected. addr points to the corrupted data, size is its size in number of bytes (1 or 2) and data is the possibly corrupted data returned from the ecc_xread function. ecc_errorhandler is called in a fatal situation, and it would probably move the system into a fail-safe mode or do a re-initialization.

```
unsigned int comp_detectederrors ;
```

```
unsigned int ecc_detectederrors ;
unsigned int ecc_fixederrors ;
```

These three global variables count the number of complement read errors and the number of fixed (1 bit) and detected (2 bit or 3 bit) ECC-errors. They are cleared by stl_init and may be checked or cleared by the user program at any time. They are incremented if a complement error was detected or if a 1 bit error was fixed (inc. ecc_fixederrors) or if a 2 bit or 3 bit ECC-error was detected (inc. ecc_detectederrors). The maximum positive value is not further incremented so that overflows are prevented.

5.4 The Application Programmer's Interface for 78K0

NEC provides a number of functions which support system self test. They may be called once at system startup, but may also be called cyclically when the system is idle.

This chapter gives a detailed description of the individual library functions.

unsigned char comp_bread(COMPBYTE *addr)
unsigned int comp_wread(COMPWORD *addr)

Execution time (max.)	67 cycles @ Byte 123 cycles @ Word
Stack requirement	2 Byte @ CompByte 4 Byte @ CompWord

Description:

Read unsigned character/word value and check for error.

The complement read functions read the data value from the specified address and verify it with its complement. The comp_errorhandler function is called, if a mismatch was detected. The error handler may decide to move the system into a fail-safe mode or it may return without further action. Wrong data may be returned in that case. The software may check the global error counter comp_detectederrors from time to time and take suitable countermeasures in case of errors. See the chapter on redundant data storage for details.

It is recommended to disable Interrupts during the read operation, so that data integrity is preserved at all times. A proposal is done in the source but commented out . Note however, that a reset can occur at any time and may therefore corrupt the data structure.

The complement read functions are not limited to RAM but they can also be performed on ROM data. Make sure to generate the proper data structure manually.

Returns:
checked data

See also:
ecc_bwrite; ecc_wwrite; ecc_bread; ecc_wread; comp_bwrite; comp_bwrite; comp_wwrite;

Example:

```
#include "stl.h"
COMPWORD val1, val2 ;
   comp_detectederrors = 0 ; /* clear global error counter */
   comp_wwrite(&val1, 1440) ;
   comp_wwrite(&val2, 311) ;
   printf("%u + %u = %u\n" ,comp_wread(&val1) ,comp_wread(&val2),
   comp_wread(&val1)+comp_wread(&val2)) ;
    if (comp_detectederrors != 0)
   { … anything to recover from detected but unfixed errors }
```

COMPBYTE *comp_bwrite(COMPBYTE *addr, unsigned char data)
COMPWORD *comp_wwrite(COMPWORD *addr, unsigned int data)

Execution time (max.)	47 cycles @ Byte 61 cycles @ Word
Stack requirement	2 Byte @ CompByte 2 Byte @ CompWord

Description:

Write unsigned character/word value with complement.

The complement write functions write a data value and its complement to the specified address. See the chapter on redundant data storage for details. It is recommended to disable Interrupts during the write operation, so that data integrity is preserved at all times. A proposal is done in the source but commented out. Note however, that a reset can occur at any time and may therefore corrupt the data structure.

Returns:
Address of the complement data structure.

See also:
ecc_bwrite; ecc_wwrite; ecc_bread; ecc_wread; comp_bread; comp_wread;

Example:

```
#include "stl.h"
COMPWORD val1, val2 ;
   comp_detectederrors = 0 ; /* clear global error counter */
   comp_wwrite(&val1, 1440) ;
   comp_wwrite(&val2, 311) ;
   printf("%u + %u = %u\n",comp_wread(&val1),comp_wread(&val2),
          comp_wread(&val1)+comp_wread(&val2)) ;
   if (comp_detectederrors != 0)
   { … anything to recover from detected but unfixed errors }
```

unsigned char ecc_bread(ECCBYTE *addr)
unsigned int ecc_wread(ECCWORD *addr)

Execution time (max.)	317 cycles @ Byte 519 cycles @ Word
Stack requirement	8 Byte @ EccByte 8 Byte @ EccWord
Resources	ecc_detectederrors ecc_fixederrors ecc_errorhandler()

Description:

Read unsigned character/word value and correct any possible error.

The ECC read functions read the data value from the specified address and detect or correct any possible error. See the chapter on redundant data storage for details. If an error is corrected, then the corrected data is written back to the memory .

If an uncorrectable read error occurs, then the ecc_errorhandler function will be called. That function may or may not return. A memory error is probably considered fatal and the system is either switched into a fail-safe mode or it is restarted.

The ECC read functions are not limited to RAM but they can also be performed on ROM data. Make sure to generate the correct redundant data either manually using the table at page 14 for encoding or running the ECC write function on the emulator or simulator.

Returns:
Corrected data.

See also:
ecc_bread; ecc_wread; comp_bwrite; comp_wwrite; comp_bread; comp_wread;

Example:

```
#include "stl.h"
ECCWORD val1, val2 ;
   ecc_detectederrors = 0 ; /* clear global ecc error counter */
   ecc_fixederrors = 0 ; /* clear global ecc error counter */
   ecc_wwrite(&val1, 1440) ;
   ecc_wwrite(&val2, 311) ;
   printf("%u + %u = %u\n" ,ecc_wread(&val1) ,ecc_wread(&val2),
   ecc_wread(&val1)+ecc_wread(&val2)) ;
   if (ecc_detectederrors != 0)
   { … anything to recover from detected but unfixed errors }
   if (ecc_fixederrors != 0)
  { … errors were found and fixed. usually uncritical, but system is in danger }
```

ECCBYTE *ecc_bwrite(ECCBYTE *addr, unsigned char data)
ECCWORD *ecc_wwrite(ECCWORD *addr, unsigned int data)

Execution time (max.)	137 cycles @ Byte 197 cycles @ Word
Stack requirement	6 Byte @ EccByte 6 Byte @ EccWord

Description:

Write unsigned character/word value with error correction code.

The ECC write function writes a data value to the specified address along with an error correction code. See the chapter on redundant data storage for details.

It is recommende to disable Interrupts during the write operation, so that data integrity is preserved at all times. A proposal is done in the source code but commented out. Note however, that a reset can occur at any time and may therefore corrupt the data structure.

Returns:
Address of the ECC data structure.

See also:
ecc_bread; ecc_wread; comp_bwrite; comp_wwrite; comp_bread; comp_wread;

Example:

```
#include "stl.h"
ECCWORD val1, val2 ;
   ecc_detectederrors = 0 ; /* clear global ecc error counter */
   ecc_fixederrors = 0 ; /* clear global ecc error counter */
   ecc_wwrite(&val1, 1440) ;
   ecc_wwrite(&val2, 311) ;
   printf("%u + %u = %u\n",ecc_wread(&val1),ecc_wread(&val2),
   ecc_wread(&val1)+ecc_wread(&val2)) ;
   if (ecc_detectederrors != 0)
   { … anything to recover from detected but unfixed errors }
   if (ecc_fixederrors != 0)
  { … errors were found and fixed. usually uncritical, but system is in danger }
```

unsigned char stl_checksum(unsigned char *addr, int size)

Execution time (max.)	615 cycles @ size = 0x0010
Stack requirement	6 Byte

Description:

Generate checksum of the data starting at **addr* with a length of *size* bytes. All bytes starting at the specified address are accumulated and overflows into higher bytes are truncated.

This is a quick test for checking the ROM integrity, but it is not as reliable as a CRC test, because multiple errors could compensate each other. To make it more reliable and to keep the execution time even smaller, it may be a good idea to divide the whole ROM space into smaller chunks and test one such chunk at a time.

Note that the NEC on-chip flash memory always uses a hardware error correction code (ECC). 38 bits are used to store a 32-bit word. Therefore this function may not be needed when using flash memory.

Returns:
checksum

See also:
stl_ram_cbtest, stl_ram_marchcm, stl_ram_marchx, stl_crc

Example:

```
#include "stl.h"
#define START_OF_ROM 0x0 /* start of ROM space for which checksums are tested */
#define CHUNK_SIZE 256   /* size of one chunk in bytes */
#define NUMBER_OF_CHUNKS 256
extern unsigned char *checksum_table ;
/* table with checksums of the individual chunks */
/* the following function is called at defined intervals */
int my_romtest(void)
{
static int noc = 0 ;
static unsigned char *taddr ;
static unsigned char *tcs ;
int rc ;
   if (noc == 256) noc = 0 ;
   if (noc == 0)
   {
      noc = NUMBER_OF_CHUNKS ;
      taddr = START_OF_ROM ;
      tcs = checksum_table ;
   }
   rc = (*tcs == stl_checksum(taddr, CHUNK_SIZE)) ;
   noc-- ;
   taddr += CHUNK_SIZE ;
   tcs++ ;
   return(rc) ;
}
```

BOOL stl_clocktest(void)

Execution time (max.)	application specific
Stack requirement	9 Byte

Description:

stl_clocktest is used to verify that the CPU main clock is still operating properly. That is a task which is highly system dependent and therefore the self test library provides only the framework to implement user specific code. In cases where an independent clock source is available, it is suggested to generate interrupts with that clock and use the stl_inttest function to verify proper operation of the CPU clock.

stl_clocktest is meant to implement timing functions, which are by their nature not very precise. The IEC60730 standard requires that only harmonics and sub-harmonics need to be tested. That means that a 50 percent error in the timing measurement can be accepted. Such low precision can be achieved with low cost RC networks or discrete RC- or ring oscillators.

Many different circuit constellations are possible. As an example, we show the idea of a simple RC timer:

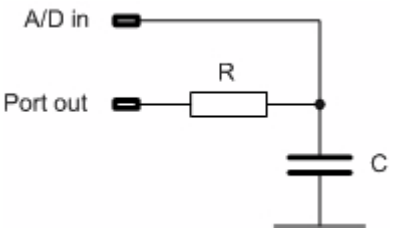

Figure 5-2 RC Timer

This timer requires a digital output port and an A/D converter input. If the A/D input is shared with a digital output port, then this single pin would be sufficient for timing measurement. The timing is simply measured by charging and discharging the capacitor C through resistor R and observing the voltage on the capacitor. As fully charging and discharging takes an infinitely long time, one can simply measure the time from 0 or V_{DD} to $V_{DD}/2$. The ADC input resistance must be high compared to resistor R and the capacitor should be high quality with no significant ageing. If a single port pin is used, the port driver must be switched to high impedance while the voltage is measured. When charging or discharging to VDD/2, the equation for the charge or discharge time is simplified to:

$$t = 0.7 * R * C$$

R should be small compared to the port input impedance. Reasonable values are 1 kΩ and 10 nF, which results in a charge and discharge time of 7 µs. That is probably long enough to provide sufficient resolution in time and short enough to be measured in an interrupt handler. Always make sure that temperature and age drifts are kept within the permitted limits.

Another possibility is to use a square wave signal connected to a port pin and measuring its pulse width. That square wave must be generated independently from the CPU clock. On-chip ring or RTC oscillators or external RC oscillators may be a good low cost choice to provide such a reference signal. Monoflops may be used as well. The stl_clocktest is a sample code to show how the CPU clock may be tested, when such a reference square wave is provided to a port pin.

The stl_clocktest sample code works by counting the number of executed instructions during a period of time, which is established by an independent clock source. Such algorithms had often been implemented on early DOS PCs and they were a pain in the neck, when faster PCs became available. However, they worked very well as long as the architecture and the clock speed remained unchanged. It seems reasonable to implement such algorithms into embedded systems, which are under full control of one manufacturer. But always keep in mind that they have to be adapted when a CPU with a different performance is selected or when the code is stored in a different memory with different access times. Otherwise the test may fail.

For these reasons, the stl_clocktest code will probably never run unchanged. It is meant as a sample, which has to be adapted to the specific needs of the application. It may not even be usable at all and there may be simpler methods in a specific system.

The implemented example is running on 78K0/KF2. The internal low speed oscillator is generating together with timer H1 a 20msec square wave signal at TOH1 output pin. 20msec are usually available on applications running on the mains. This signal is then the inputsignal to port P15. The number of instructions executed during the high period (10msec) are counted, which gives a clear indication whether the CPU is running on it’s fundamental clock or an harmonic. (for details see stl_clocktest).

Returns:
0 if the CPU clock is operating within its limits, !=0 on any error

See also:
stl_inttest

Example:

```
#include “stl.h”
   if (stl_clocktest() != 0)
   {
      error handler
   }
   else
   {
      proceed normally
   }
```

unsigned int stl_crc(unsigned int crc, void *data, unsigned int length)

Execution time (max.)	31749 cycles @ length = 0x00FF
Stack requirement	4 Byte

Description:

stl_crc calculates the cyclic redundancy check (CRC) of the data starting at **data* in the length of *length* bytes. The CRC algorithm chosen is CRC16. *crc* is the initial CRC and stl_crc will return the new CRC of the specified data string. CRC calculation for longer strings may be split into shorter ones by specifying the previously calculated one as the initial CRC for the subsequent call to stl_crc. That keeps individual runtimes short. The initial CRC when starting a new CRC check is often 0, but sometimes other values like -1 or 0x1d0f are used. Make sure to use the same initial value for generating and checking a CRC.

Returns:
CRC over *length* bytes of data starting at **data*.

See also:
stl_checksum

Example:

```
#include "stl.h"
unsigned int crc ;
unsigned char data[] = "1234567890abcdefghijklmnopqrstuvwxyz" ;
   crc = stl_crc(0, data, sizeof(data) - 1) ;
```

BOOL stl_init(void)

Execution time (max.)	65 cycles
Stack requirement	2 Byte
Resources	all global variables

Description:

stl_init initializes the self test library. It must be called once before using any of the stl functions.

Returns:
0 if the library was successfully initialized, !=0 on any error

See also:

Example:

```
#include "stl.h"
   if (stl_init() != 0)
   {
      error handler
   }
   else
   {
      proceed normally
   }
```

BOOL stl_inttest(INTCHK *chk)

Execution time (max.)	System Dependent
Stack requirement	System Dependent

Description:

stl_inttest is used to verify that CPU interrupts are handled in time. That is a task which is highly system dependent and therefore the self test library can only contribute the wrap up handler, which checks that a number of specific interrupts occurred at least and at most a predefined number of times.

It is assumed that the self test functions including stl_inttest are called in specified intervals, e.g. triggered by a timer or line frequency interrupt. Depending on the system, typical intervals would probably be between 10 ms and 1000 ms. Other system interrupts, which shall be verified by this function, may occur in vastly different intervals. They may occur more often or much more rarely, sometimes even never.

The principle function of stl_inttest is frequency measurement. This is achieved by counting the number of interrupts between calls to the stl_inttest function. Each specific interrupt handler which is to be supervised, must decrement a dedicated global variable. stl_inttest compares that variable to predefined upper and lower bounds, sets it to its preset value and returns an error, if the limits are exceeded. The variable is unsigned and must therefore not be decremented below zero.

For such interrupts, which do not occur often enough, stl_inttest will decrement a tick counter if no interrupt occurred during the last interval. Otherwise the tick counter will be reset to its initial value. stl_inttest will return an error condition, if the tick counter is zero and its initial value is not zero or if it is decremented to zero. The tick counter is unsigned and will not be decremented if it is already zero.

Since stl_inttest requires many parameters, the data type INTCHK has been defined and a pointer to an INTCHK type is passed to this function. INTCHK is defined as follows:

```
typedef struct {
                unsigned int *freq ;
                unsigned int *ticks ;
          const unsigned int *freq_lower ;
          const unsigned int *freq_upper ;
          const unsigned int *freq_initial ;
          const unsigned int *ticks_initial ;
                  signed int array_size ;
               } INTCHK ;
```

freq is a pointer to an array of frequency counters, each of which is decremented by its associated interrupt handler. ticks is a pointer to an array of tick counters, which are decremented when the frequency counters were not updated since the previous call. freq_lower and freq_upper are arrays of the same size and specify the upper and lower permitted frequencies respectively. freq_initial is the initial value for the frequency counters and ticks_initial is the initial value for the ticks counters. Both will be used by stl_inttest to initialize the respective counters. array_size is the size of the arrays. The frequency and tick counters must obviously be located in RAM. The other arrays are constant and may be located in ROM.

The picture below shows the implementation. stl_inttest shall be called in regular intervals, e.g. triggered by the 50 Hz line frequency. int is the interrupt from the supervised interrupt-input, whose handler decrements the counter freq. stl_inttest generates an error code, if the counter is not within its predefined limits, i.e. if there were too many or too few interrupts.

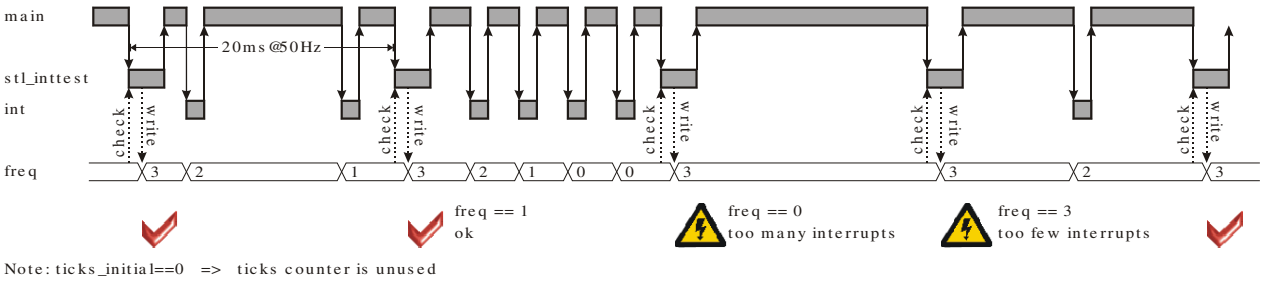

Figure 5-3 implementation stl_inttest

The picture below shows how the ticks counter works. The ticks counter is decremented by stl_inttest, if the associated interrupt did not occur during the previous interval. Otherwise it is reset to its initial value, in this case 5. stl_inttest will return an error condition, if the ticks counter is zero and if its initial value is not zero. For frequent interrupts, if the ticks counter is not needed, ticks_initial shall be 0, which effectively disables the ticks algorithm.

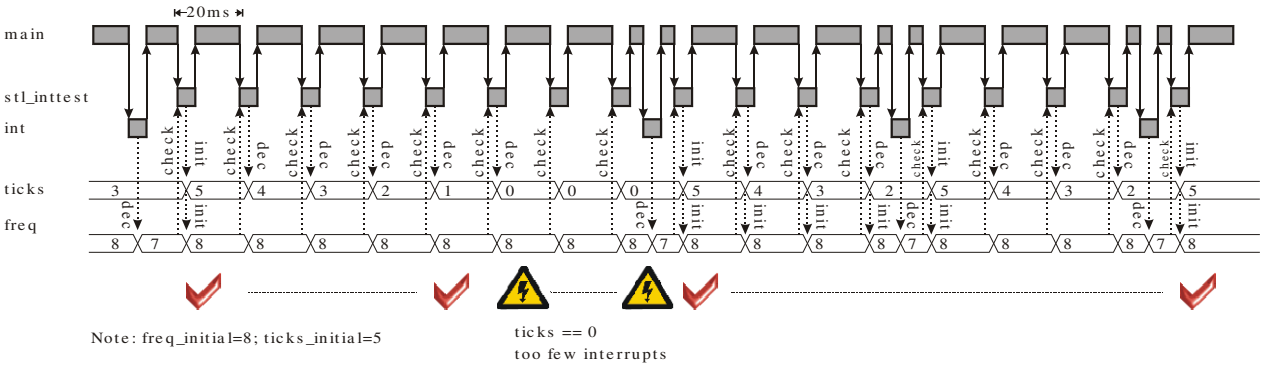

Figure 5-4 tick counter operation

stl_inttest should also be used to verify the clock speed of the CPU. The IEC60730 standard requires that only harmonics and sub-harmonics need to be tested. If an independent clock is available (e.g. line frequency, ring oscillator, low cost RC oscillators or real time clock), it may be used to generate interrupts and these interrupt intervals may be verified by stl_inttest as described above.

Returns:
0 if the interrupt test was successful, !=0 on any error

See also:
stl_clocktest

Example:

```
#include "stl.h"
unsigned int freq[4] ;
unsigned int ticks[4] ;
const unsigned int freq_lower[] = {75, 10, 0, 4} ;
const unsigned int freq_upper[] = {120, 20, 300, 5} ;
const unsigned int freq_initial[] = {1000, 1000, 300, 40} ;
const unsigned int ticks_initial[] = {0, 0, 10, 0} ;
INTCHK intchk = {freq, ticks, freq_lower, freq_upper, freq_initial,
ticks_initial, sizeof(freq)/sizeof(freq[0])} ;
// The following code is executed periodically, e.g. every 50ms.
// For demonstration, the individual calls are lined up below and
// the interrupts are simulated by function calls.
// call stl_inttest once to initialize all counters. We ignore
// the return value for this first call.
   stl_inttest(&intchk) ;
   int0(80) ; int1(15) ; int2(150) ; int3(5) ; //simulate int's
   error = stl_inttest(&intchk) ; // no error, all counters
```

```
                                    // within bounds
   int0(74) ; int1(20) ; int2(1) ; int3(4) ; //simulate int's
   error = stl_inttest(&intchk) ; // error, counter 0 is
                                    // below lower bound
   for (i=0;i<9;i++)
   {
      int0(80) ; int1(15) ; int2(0) ; int3(5) ; //simulate int's
      error = stl_inttest(&intchk) ; // no error, all counters
                                      // within bounds
   }
   int0(80) ; int1(15) ; int2(0) ; int3(5) ; //simulate int's
   error = stl_inttest(&intchk) ; // error, ticks counter 2
                                    // decrements to zero
Interrupt handler n:
   …
if (intchk.freq[n]!=0) intchk.freq[n]-- ; /* interrupts must be disabled */
   …
```

BOOL stl_pctest(void)

Execution time (max.)	366 cycles @ 11 PC jumps
Stack requirement	10 Byte
Resources	stl_pc_const.xcl

Description:

The *stl_pctest* function performs a functional test of the program counter (PC). That is achieved by branching to multiple code snippets at different locations in main memory. These snippets generate a unique return value, which are verified by stl_pctest.

The number of code snippets to be generated can be defined, editing the stl_pctest funktion. PCTEST1 to PCTESTB are implemented in the 78K0 example. The number of code snippets is equal to the number of PCTESTx. The locations/ addresses were the PC jumps to is defined in stl_pc_const.xcl.

Choosing the proper locations for the individual code snippets depends on the memory map of the system. It is a good idea to spread the code across all memories, so that each memory block is tested at least once.

Returns:
0 if no fault was detected, !=0 on any error

See also:

Example:

```
#include "stl.h"
int selftest()
{
int rc ;
   rc = 0 ;
   if (stl_pctest() != 0) rc = 1 ;
   … (other self tests)
   return(rc) ;
}
```

BOOL stl_ram_marchcmb(unsigned char *addr, int num)

Execution time (max.)	4931 cycles @ num = 0x0005
Stack requirement	10 Byte

Description:

stl_ram_marchcm test performs a March C- test on the memory starting at address *addr* with a length of *num* bytes. The March C- test is rather comprehensive, as it finds all stuck-at, addressing, transition and coupling faults. stl_ram_marchcm is destructive, i.e. the memory contents is not preserved. Therefore it is meant to run at system startup, before the memory and the run time library are initialized. The memory will be cleared (=0) when stl_ram_marchcm returns.

As the March C- test is rather slow, one might consider splitting it up and doing multiple runs on smaller chunks of memory. That would, however, not decrease the total run time, but it would reduce its capability to find address decoder faults. Therefore it is recommended to run the March C- test always on the complete memory.

stl_ram_marchcm uses C calling conventions, so it may be called from C during system operation. That may be of limited use, because most of the memory is probably initialized and must not be destroyed. Running a March test on a subset of the total memory area may not find all possible memory errors. Some address decoder faults may remain undetected.

stl_ram_marchcmb uses byte accesses. The address must be properly aligned to the data type and the length must be an integral multiple of the data width.

Returns:
0 if no memory fault was detected, -1 otherwise.

See also:
stl_ram_cbtest, stl_ram_marchx

Example:

```
#include "stl.h"
#define MARCHCMW_START ((unsigned int *)0x100000) //start address
#define MARCHCMW_LEN   0x2000                     //length in byte
int rc ;
   …
   rc = stl_ram_marchcmw(MARCHCMW_START, MARCHCMW_LEN) ;
   …
}
```

BOOL stl_ram_marchxb(unsigned char *addr, int num)

Execution time (max.)	2739 cycles @ num = 0x0005
Stack requirement	10 Byte

Description:

stl_ram_marchx performs a March X test on the memory starting at address *addr* with a length of *num* bytes. The March X test is similar to the March C- test, but it executes quicker and does not find all coupling faults. Like March C- it finds all stuck-at, addressing and transition faults. stl_ram_marchx is destructive, i.e. the memory contents is not preserved. Therefore it is meant to run at system startup, before the memory and the run time library are initialized. The memory will be cleared (=0) when stl_ram_marchx returns.

As the March X test is rather slow, one might consider splitting it up and doing multiple runs on smaller chunks of memory. That would, however, not decrease the total run time, but it would reduce its capability to find address decoder faults. Therefore it is recommended to run the March X test always on the complete memory.

stl_ram_marchx uses C calling conventions, so it may be called from C during system operation. That may be of limited use, because most of the memory is probably initialized and must not be destroyed. Running a March test on a subset of the total memory area may not find all possible memory errors. Some address decoder faults may remain undetected.

stl_ram_marchxmb uses byte accesses.The address must be properly aligned to the data type and the length must be an integral multiple of the data width

Returns:
0 if no memory fault was detected, -1 otherwise.

See also:
stl_ram_cbtest, stl_ram_marchcm

Example:

```
#include “stl.h”
#define MARCHXW_START ((unsigned int *) 0x100000) //start address
#define MARCHXW_LEN   0x2000                      //length in byte
int rc ;
   …
   rc = stl_ram_marchxw(MARCHXW_START, MARCHXW_LEN) ;
   …
}
```

BOOL *stl_ram_cbtestb(unsigned char *addr, int num)

Execution time (max.)	645 cycles @ num = 0x0006
Stack requirement	10 Byte

Description:

stl_ram_cbtest is a checkerboard test for *num* bytes of the RAM block starting at address *addr*. The memory data is preserved. As memory busses often retain the previously written data due to the bus capacitance, even if no physical memory is connected, the pattern test is performed in data pairs. stl_ram_cbtest saves the current memory contents and then writes a checkerboard pattern to the memory location at address n and the inverted pattern to n+1. For this reason, the number of elements must be even. stl_ram_cbtest verifies the contents, writes inverted pattern to the same locations and verifies again before it restores the original values.

It is recommende to disabled Interrupts during each 2 Byte RAM test operation, so that data integrity is preserved at all times. A proposal is done in the source but commented out . Note however, that a reset can occur at any time and may therefore corrupt the data structure

As the checkerboard test modifies the memory contents, servicing of non-maskable interrupts and CPU resets require special care. A CPU reset may occur at any time and therefore the tested memory contents may be left in a corrupted state, which cannot be avoided or fixed easily.

stl_ram_cbtestb has been implemented, which uses byte accesses. The address must be properly aligned to the data type and the length must be an integral multiple of twice the data width. For performance reasons the access width should match the bus width of the memory block under test.

Returns:
0 if the RAM block is found to be correct, !=0 on any error

See also:
stl_ram_marchcm, stl_ram_marchx

Example:

```
#include "stl.h"
#define START_OF_RAM 0x1000 /* start of RAM space to be tested */
#define CHUNK_SIZE 256      /* size of one chunk in elements */
#define NUMBER_OF_CHUNKS 256
/* the following function is called at defined intervals */
int my_ramtest(void)
{
static int noc = 0 ;
static unsigned char *taddr ;
int rc ;
   if (noc == 0)
   {
      noc = NUMBER_OF_CHUNKS ;
      taddr = START_OF_RAM ;
   }
   rc = stl_ram_cbtests(taddr, CHUNK_SIZE) ;
   noc-- ;
   taddr += CHUNK_SIZE ;
   return(rc) ;
}
```

BOOL stl_registertest(void)

Execution time (max.)	1003 cycles
Stack requirement	18 Byte

Description:

The *stl_registertest* function performs a functional test of the CPU core registers. That includes all general purpose registers (AX, BC, DE, HL) on all four banks. All tests are non destructive, i.e. all registers are saved on the stack and restored before the function returns. Only the register, which conveys the return code, is destroyed.

It is recommende to disabled Interrupts during the register test, so that data integrity is preserved at all times. A proposal is done in the source but commented out .

The registers are tested in pairs by loading one of them with 0xaa and the other one with 0x55. Then the first register contents is read and compared to 0xaa and the second register is read and compared to 0x55. The function will return with an error code, if the values do not match. This function will detect single bit stuck at 0 and stuck at 1 errors.

Returns:
0 if no fault was detected, !=0 on any error

See also:

Example:

```
#include "stl.h"
int selftest()
{
int rc ;
   rc = 0 ;
   if (stl_registertest() != 0) rc = 1 ;
   … (other self tests)
   return(rc) ;
}
```

Chapter 6 Flowcharts

stl_checksum		
Function Call from C	unsigned char stl_checksum (unsigned char *addr, int size)	
start address checksum	signed char * addr	> register AX
number of bytes	int size	> register BC
return value	calculated checksum	< register A

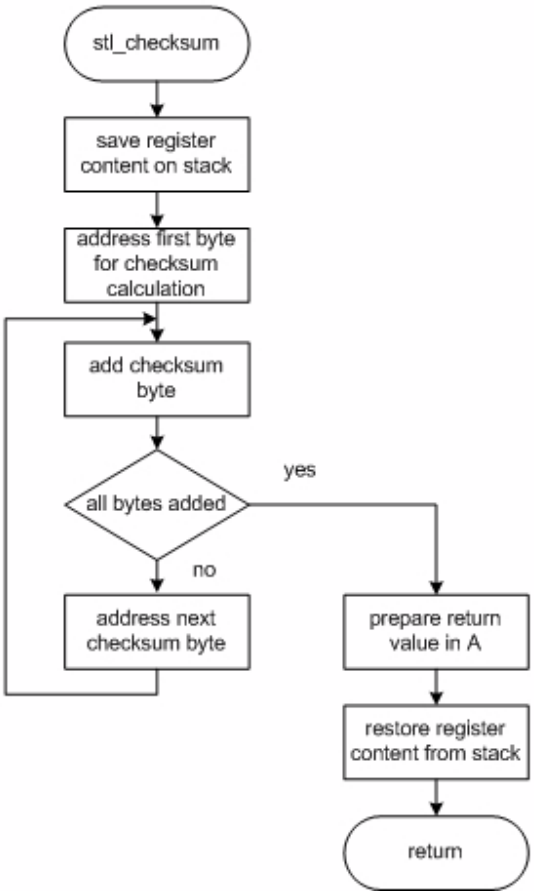

Figure 6-5 Checksum test

comp_bread & comp_wread		
Function Call from C	unsigned char comp_bread (COMPBYTE *addr) unsigned char comp_wread (COMPWORD *addr)	
address of original/unencoded data	COMPBYTE *addr COMPWORD *addr	> register AX > register AX
return value	complement checked data	< register A

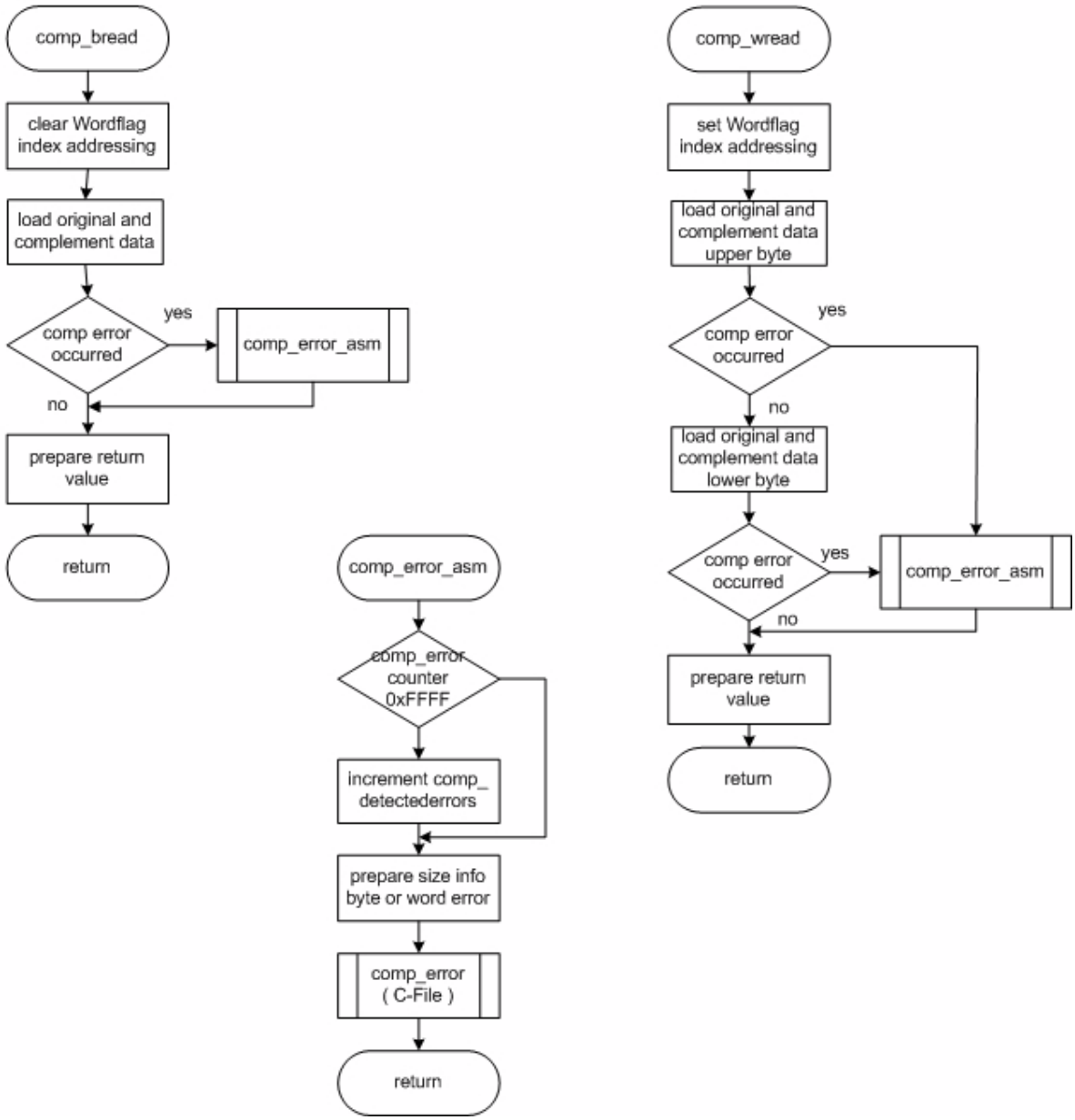

Figure 6-6 Comp test – read

Comp_test		
Function Prototype	unsigned char COMPBYTE *comp_bwrite (COMPBYTE *addr, unsigned char data) unsigned char COMPWORD *comp_wwrite (COMPWORD *addr, unsigned int data)	
address of data	COMPBYTE *addr	> register AX
	COMPWORD *addr	> register AX
data into comp_xwrite	unsigned char data	> register B
	unsigned int data	> register BC
return value	non	

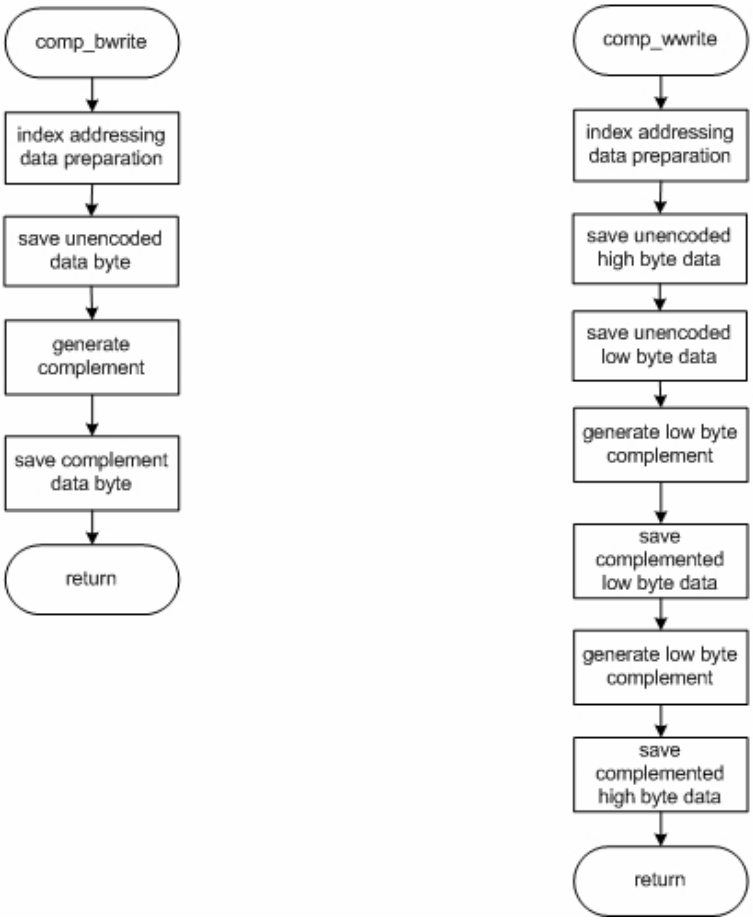

Figure 6-7 Comp test - write

stl_clocktest		
Function Call from C	bool stl_clocktest (void)	
return value	clock test result	< register A

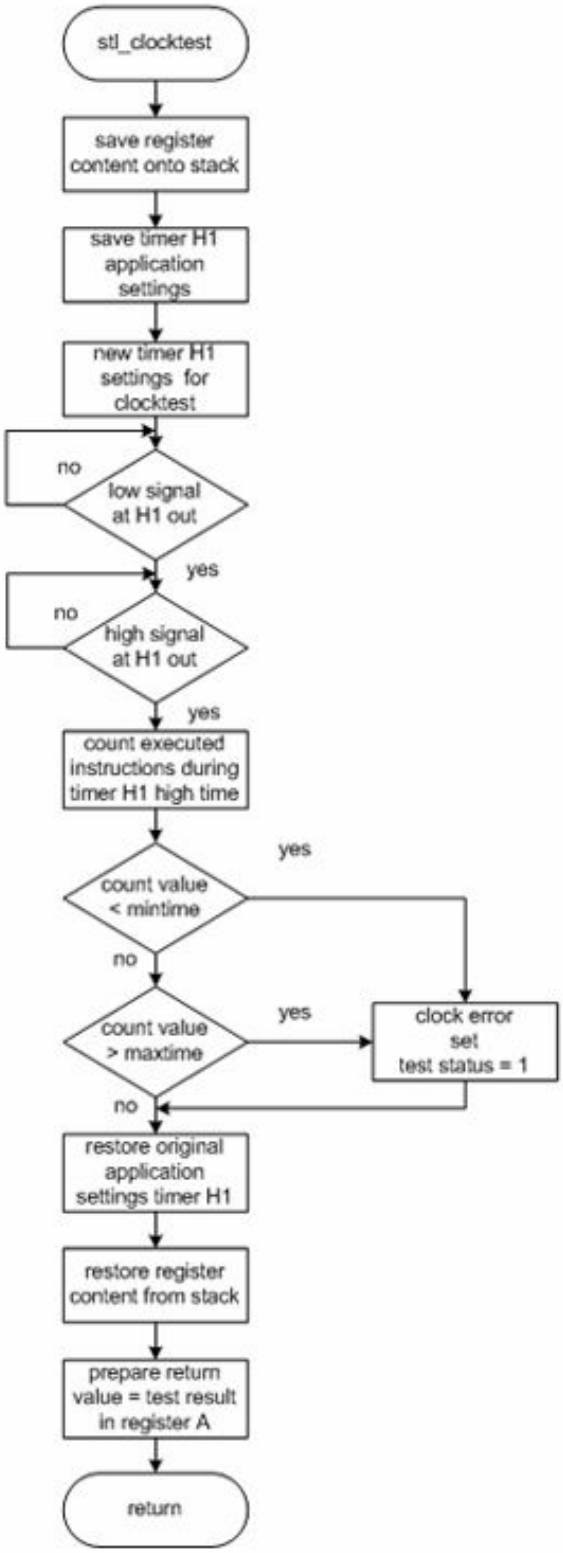

Figure 6-8 Clocktest

ecc_bwrite & ecc_wwrite		
Function Call from C	ECCBYTE *ecc_bwrite (ECCBYTE *addr, unsigned char data) ECCWORD *ecc_wwrite (ECCWORD *addr, unsigned int data)	
address of encoded data	ECCBYTE *addr ECCWORD *addr	> register AX > register AX
to be encoded data	unsigned char data unsigned int data	> register B > register BC
return value	non	

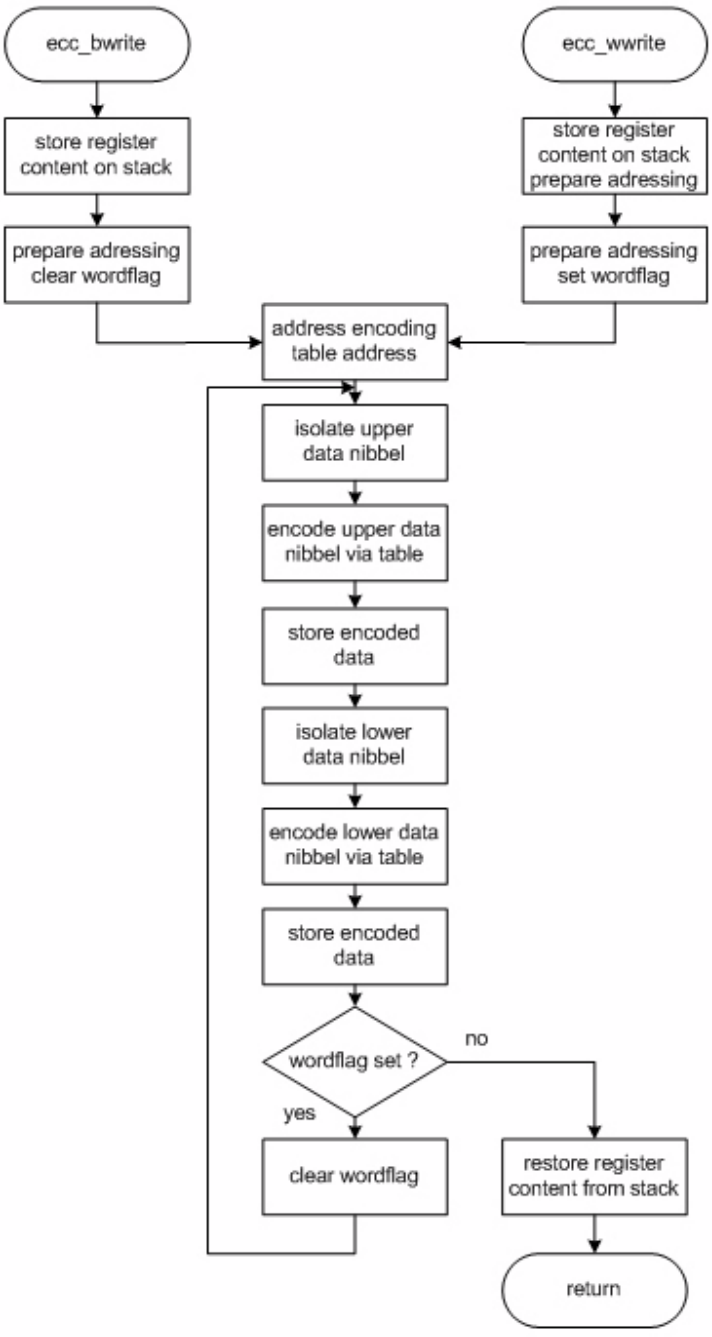

Figure 6-9 ecc test – write

ecc_bread & eccwread

Function Call from C	unsigned char ecc_bread (ECCBYTE *addr) unsigned int ecc_wread (ECCWORD *addr)	
address of encoded data	ECCBYTE *addr ECCWORD *addr	> register AX > register AX
return value	decoded data (ecc_bread) decoded data (ecc_wread)	< register A < register AX

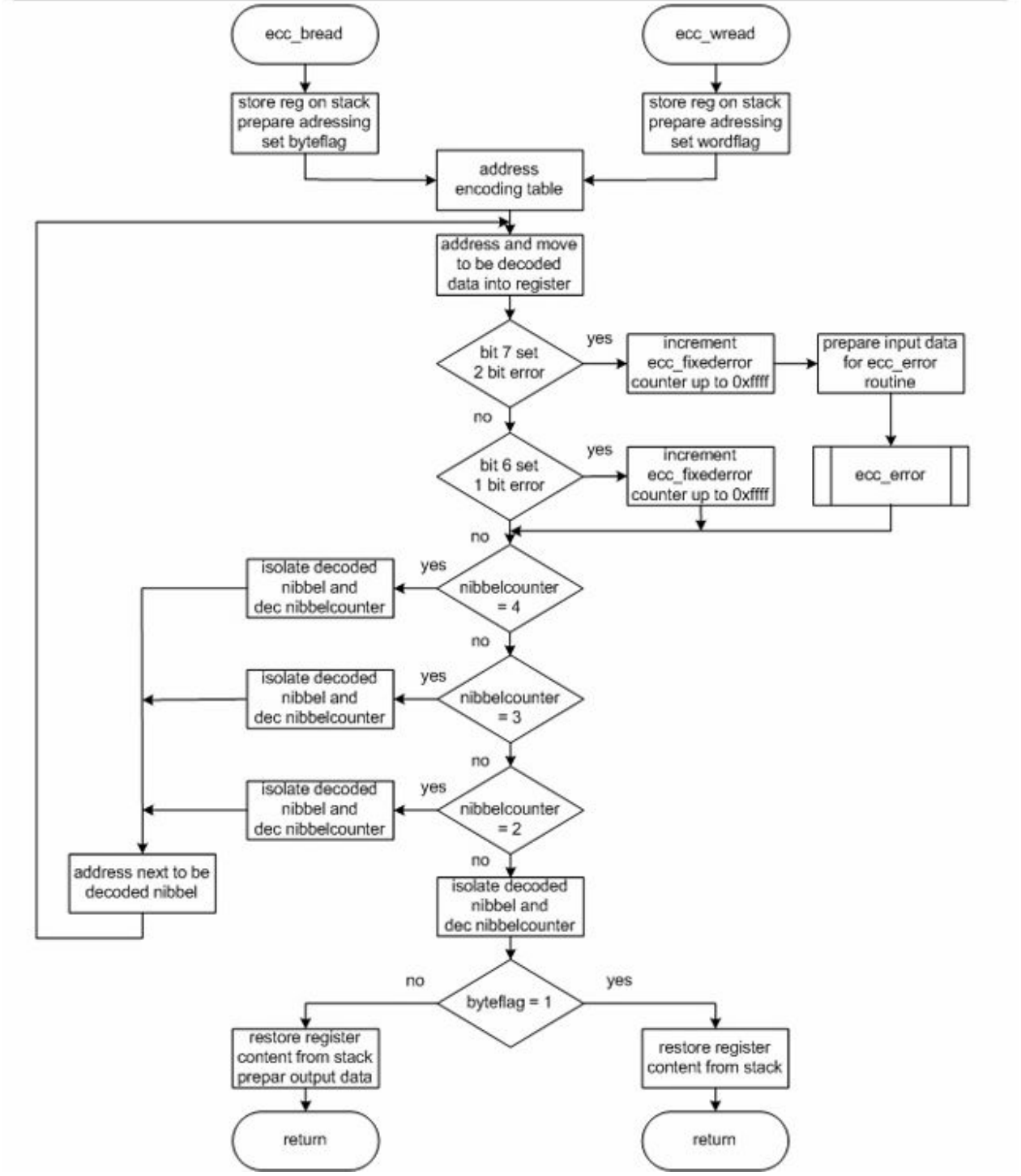

Figure 6-10 ecc test – read

stl_inttest		
Function Call from C	bool stl_inttest(INTCHK *chk)	
	INTCHK *chk	> register AX
return value	interrupt test result (BOOL)	< register A

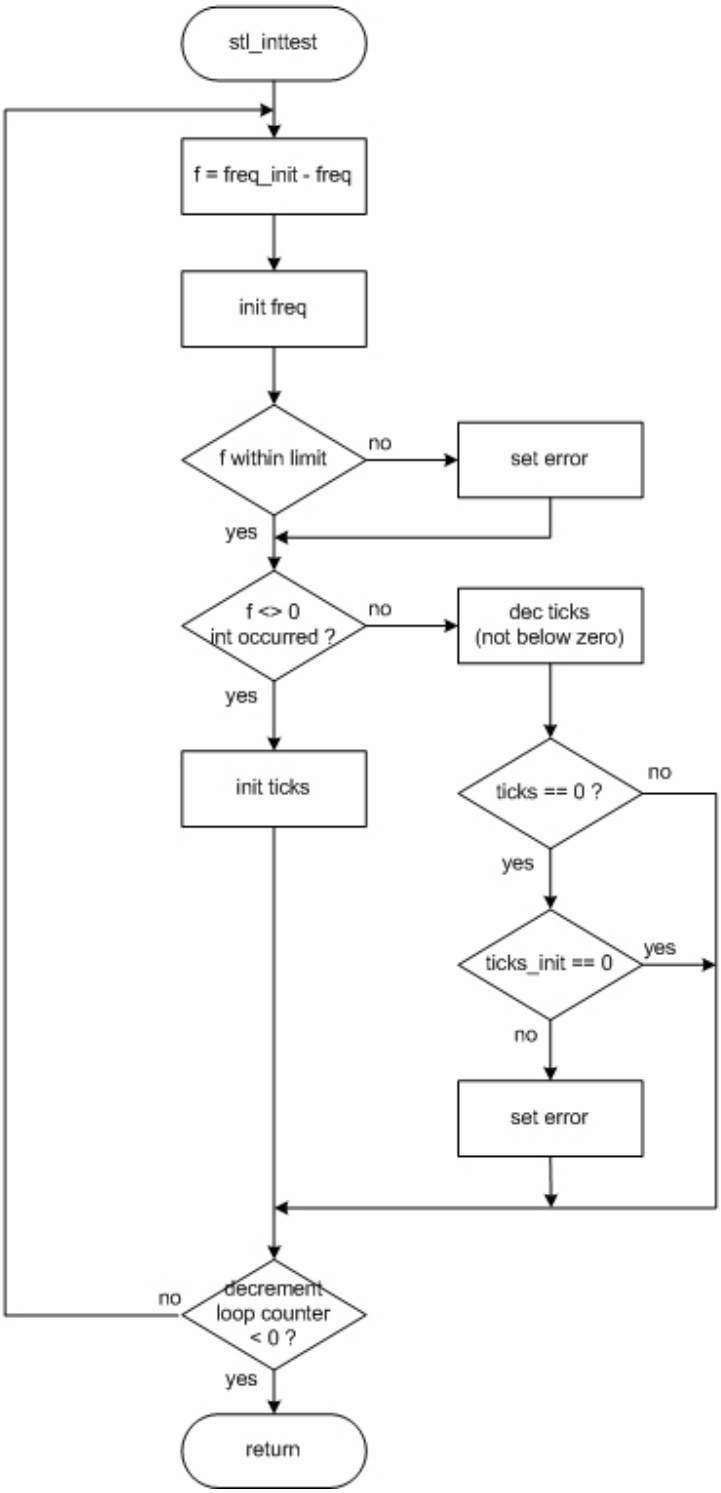

Figure 6-11 Interrupt test

stl_pctest		
Function Call from C	bool stl_pctest(void)	
return value	pctest result (BOOL)	< register A

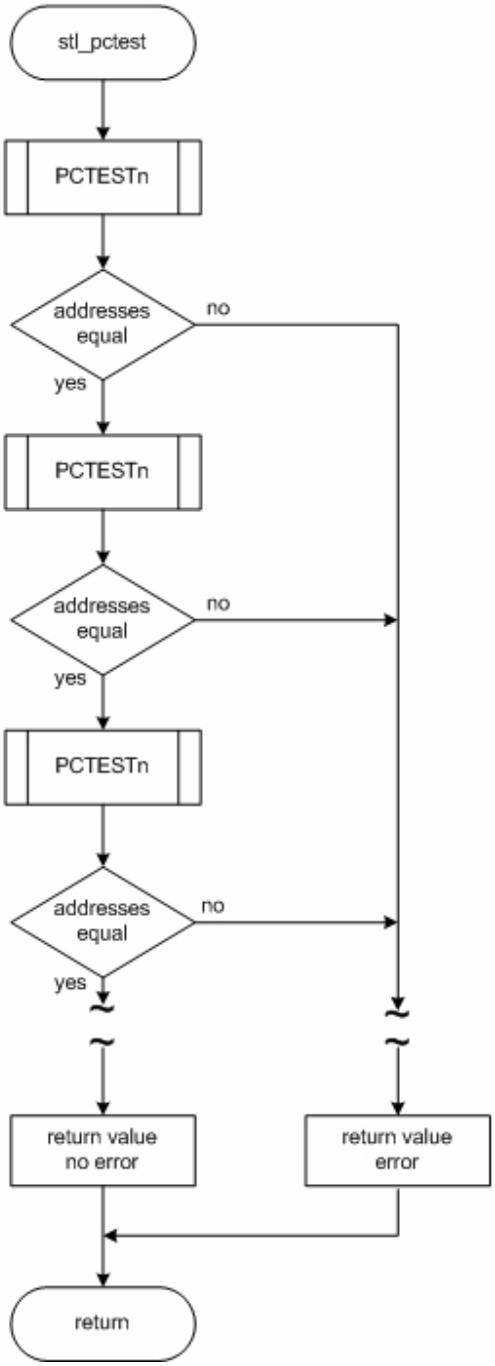

Figure 6-12 Programm counter test

stl_ram_cb_testb		
Function Call from C	BOOL stl_ram_cb_testb(unsigned char *addr, int num)	
start address checkerboard test	unsigned char *addr	> register AX
number of bytes to betested	int num	> register BC
return value	bool	< A

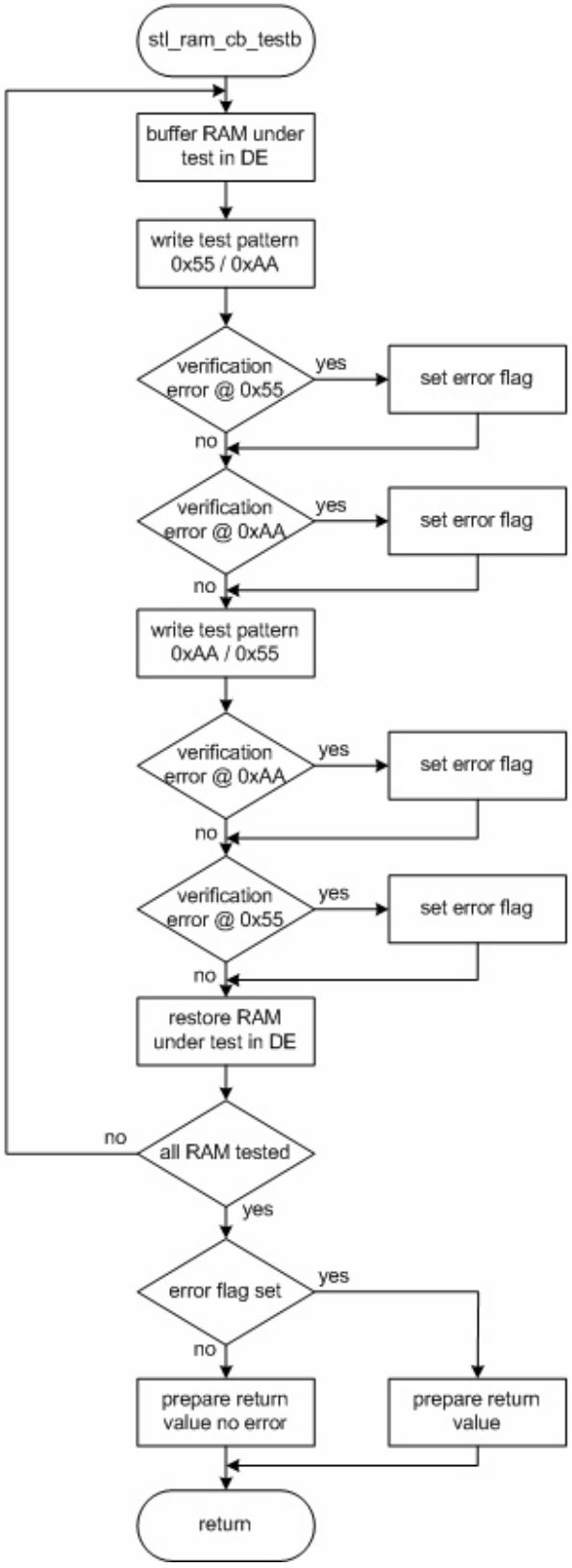

Figure 6-13 RAM checker board test

stl_ram_marchcmb		
Function Call from C	BOOL stl_ram_marchcmb(unsigned char *addr, int num)	
start address march c test	unsigned char *addr	> register AX
number of bytes to be ested	int num	> register BC
return value	bool	< A

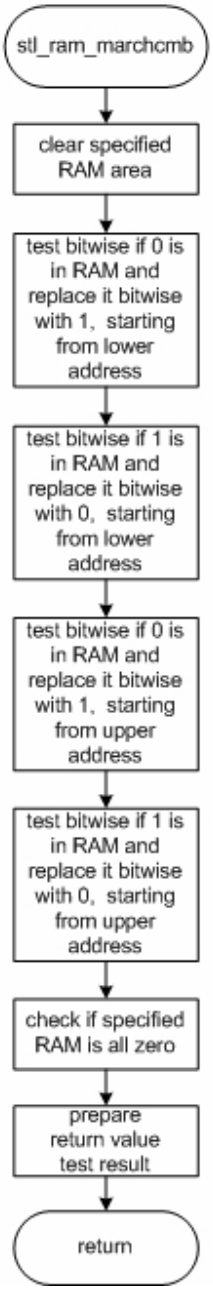

Figure 6-14 RAM march c test

stl_ram_marchxmb		
Function Call from C	BOOL stl_ram_marchxmb(unsigned char *addr, int num)	
start address march x test	unsigned char *addr	> register AX
number of bytes to be tested	int num	> register BC
return value	bool	< A

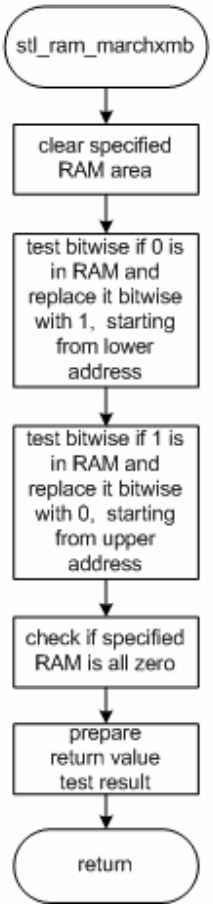

Figure 6-15 RAM march x test

stl_registertest		
Function Call from C	BOOL stl_registertest(void)	
return value	bool	< A

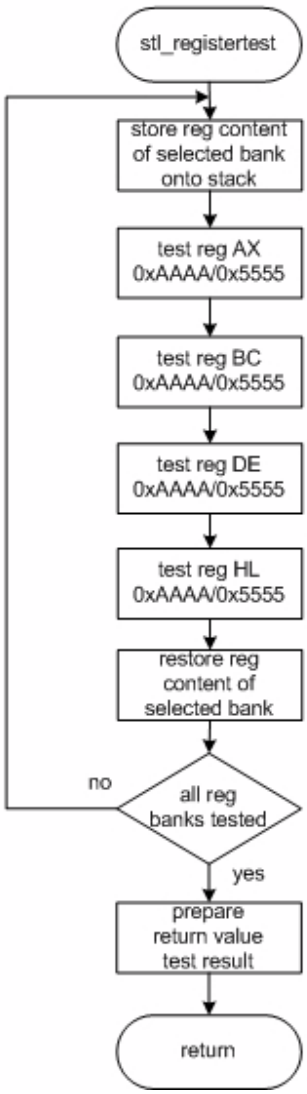

Figure 6-16 Registertest

Chapter 7 Program Listing

```
//*************************************************************************************************
//                                                                                                 *
//  Project:     78K0 Selftest                                                                     *
//  Device:      78K0/KF2 ( uPDF0547 )                                                             *
//  Main:        Endlessloop as a frame for testing/demonstrating Self Test Library for IEC 60730  *
//                                                                                                 *
//               NEC Electronis Europe                                                             *
//*************************************************************************************************

// Standard ICC78K include files
#include <stdbool.h>
#include <intrinsics.h>
#include <io78f0547_80.h>

// Projekt include files
#include <stl.h>
#include <main.h>
#include <stl_gobal_data_example.h>

// definitions interrupt example
unsigned int freq[4] ;
unsigned int ticks[4] ;
const unsigned int freq_lower[] = { 0,0,0,0 };
const unsigned int freq_upper[] = { 120, 20, 300, 8 };
const unsigned int freq_initial[] = { 1000, 1000, 300, 40 };
const unsigned int ticks_initial[] = { 0, 0, 10, 0 };


INTCHK intchk = {freq, ticks, freq_lower, freq_upper, freq_initial, ticks_initial, sizeof(freq)/sizeof(freq[0])};

void int0(unsigned int count) {if (intchk.freq[0]>count)intchk.freq[0]-=count ; else intchk.freq[0]=0 ; }
void int1(unsigned int count) {if (intchk.freq[1]>count)intchk.freq[1]-=count ; else intchk.freq[1]=0 ; }
void int2(unsigned int count) {if (intchk.freq[2]>count)intchk.freq[2]-=count ; else intchk.freq[2]=0 ; }
void int3(unsigned int count) {if (intchk.freq[3]>count)intchk.freq[3]-=count ; else intchk.freq[3]=0 ; }


//*************************************************************************************************
 int main( void )
{
  PeripheralInit();
  stl_init();

  while (1)
  {
    for (I=0; I<255; I++)                    // Increment LED PWM Timer
    {
        DelayTimer_51();
        CR50 = I;
    }

    for (I=255; I>0; I--)                    // Decrement LED PWM Timer
    {
        DelayTimer_51();
        CR50 = I;

    }

//*************************************************************************************************
// from here on each available SFT Funktion is executed using some example parameters              *
//*************************************************************************************************

//-------------------------------------------------------------------------------------------------
//  DESCRIPTION: comp_bwrite/comp_wwrite
//      writes byte/word data with it's complement into memory
//  RETURN: address of the complement data structure
//-------------------------------------------------------------------------------------------------
    comp_bwrite(&cb1,0x1A) ;
```

```
    comp_wwrite(&cw1,0x345A) ;

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: comp_bread/comp_wread
//      reads byte/word data with it's complement from memory and checks for errors
//  RETURN: checked data
//-----------------------------------------------------------------------------------------------------------
    cb = comp_bread(&cb1) ;
    cw = comp_wread(&cw1) ;

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: ecc_bwrite/ecc_wwrite
//      writes byte/word data with error correction code into memory
//  RETURN: address of complemented data structure
//-----------------------------------------------------------------------------------------------------------
    ecc_bwrite(&eb1,0xAB) ;
    ecc_wwrite(&ew1,0xDEAD) ;

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: ecc_bread/ecc_wread
//      reads byte/word datafrom memory and correct 1bit errors, indicates 2 bit errors
//      caution ! in case of 1bit error ecc_fixed errors will be incremented and return value is correct
//      caution ! in case of 2bit error ecc_detectederrors will be incremented but return value is not correct
//  RETURN: corrected data
//-----------------------------------------------------------------------------------------------------------
    eb = ecc_bread(&eb1) ;
    ew = ecc_wread(&ew1) ;

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_checksum
//      generates checksum of the data starting at *addr with a length of size bytes. All bytes starting
//      at the specified address are accumulated and any overflow is ignored
//  Return: checksum
//-----------------------------------------------------------------------------------------------------------
    addr = ( unsigned char * ) 0x0100 ;
    size = 0x0100 ;
    result = stl_checksum( addr, size );

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_clocktest
//      an example of a simple clocktest. Based on the mains, during 10msec, generated by timer H1,
//      the number of instructions executed are counted with an specified range
//  RETURN: 0 if CPU clock is operating within limits, 1 otherwise
//-----------------------------------------------------------------------------------------------------------
    resultBool = stl_clocktest();
    indicate_test_result( resultBool );

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_crc
//      this function calculates the cyclic redundancy check (CRC) of data starting at "*data" with the
//      length of "length " . "crc" is the initial CRC, and stl_crc will return the new CRC of the
//      specified datastring.
//  RETURN: CRC over "crc_length" starting at "pStAd"
//-----------------------------------------------------------------------------------------------------------
    crc = 0x0000;
    pStAdr = (unsigned char *) 0x1234;
    crc_length = 0x00FF;
    crc = stl_crc(crc, crc_length, pStAdr);

//-----------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_fast_crc16
//      this function is basically the same as stl_crc but based on C code ( just for reference and testing )
//      this function calculates the cyclic redundancy check (CRC) of data starting at "*data" with the
//      length of "length " . "crc" is the initial CRC, and stl_crc will return the new CRC of the
//      specified datastring.
//  RETURN: CRC over "crc_length" starting at "pStAd"
//-----------------------------------------------------------------------------------------------------------
    crc = 0x0000;
    pStAdr = (unsigned char *) 0x1234;
```

```
    crc_length = 0x00FF;
    crc = fast_crc16(crc, crc_length, pStAdr);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_pctest
//      this function performs a functional test of the program counter. That is achieved by branching
//      to multiple code snippets at different locations in main memory. The unique return value is
//      verified by stl_pctest (  see also: stl_pc_const.xcl )
//  RETURN: 0 if no fault was detected,  1 otherwise
//-------------------------------------------------------------------------------------------------------------
    resultBool = stl_pctest();
    indicate_test_result (resultBool);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_ram_marchcmb
//      stl_ram_marchsmb test performs a March C- test on the memory starting at address "addr" with a
//      length of "num" bytes. this test is destructive, the memory content is not presereved, therefore
//      this test could performed at startup before any system initilization.
//  RETURN: 0 if no memory fault was detected, 1 otherwise
//-------------------------------------------------------------------------------------------------------------
    p_MC_StAdr = (unsigned char *) 0xFB00;
    RAM_MarchC_lgth = 0x0005;
    resultBool = stl_ram_marchcmb(p_MC_StAdr, RAM_MarchC_lgth);
    indicate_test_result (resultBool);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_ram_marchxmb
//      stl_ram_marchsmb test performs a March X test on the memory starting at address "addr" with a
//      length of "num" bytes. this test is destructive, the memory content is not presereved, therefore
//      this test could performed at startup before any system initilization.
//  RETURN: 0 if no memory fault was detected, 1 otherwise
//-------------------------------------------------------------------------------------------------------------
    p_MX_StAdr = (unsigned char *) 0xFB05;
    RAM_MarchX_lgth = 0x0005;
    resultBool = stl_ram_marchxmb(p_MX_StAdr, RAM_MarchX_lgth);
    indicate_test_result (resultBool);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_ram_cb_testb
//      this is a checkerboard test for "num" bytes of RAM blocks, starting at address "addr"
//      the current memory content is saved, then checkerboard pattern is written to RAM location n and
//      the inverted one to n+1. For this reason, the number of elements must be even
//  RETURN: 0 if no memory fault was detected, 1 otherwise
//-------------------------------------------------------------------------------------------------------------
    addr = ( unsigned char * ) 0xFB10 ;
    num = 0x0006 ;
    resultBool = stl_ram_cb_testb( addr, num );
    indicate_test_result (resultBool);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_registertest
//      performs a functional test of the CPU core register ( AX, BC, DE, HL - all four banks ).
//      The test is non destructive, all are saved on the stack and restored before the function returns.
//  RETURN: 0 if no memory fault was detected, 1 otherwise
//-------------------------------------------------------------------------------------------------------------
    resultBool = stl_registertest();
    indicate_test_result (resultBool);

//-------------------------------------------------------------------------------------------------------------
//  DESCRIPTION: stl_inttest
//      is used to verify that CPU interrupts are handeld in time. This task is highly system dependent
//      and therefore this function can only contribute the wrap up handler, checking the number of
//      interrupts occured at least and at most in a predefined number of times.
//      As many parameters are rquiered, the datatype INTCHK has been defined and a pointer passes to that
//      function
//  RETURN: 0 if the interrupt test was successful, 1 on any error
//-------------------------------------------------------------------------------------------------------------
//    resultBool = stl_inttest(&intchk);
//-------------------------------------------------------------------------------------------------------------
```

```
//   stl_interrupt_test_excample();

//*****************************************************************************************
// interrupt test starts from here
    stl_inttest(&intchk);  // ignore result, function call is just to initilise inttest

//********settings for this test*************************************************************
/*unsigned int freq[4] ;
unsigned int ticks[4] ;
const unsigned int freq_lower[] = { 0,0,0,0 };
const unsigned int freq_upper[] = { 120, 20, 300, 8 };
const unsigned int freq_initial[] = { 1000, 1000, 300, 40 };
const unsigned int ticks_initial[] = { 0, 0, 10, 0 };
*/


// test stl_inttest (no ticks included )
// simulate counted interrupts by putting fixed values into "freq" counter
    int0(150);      // error, because above upper limit
    int1(10);
    int2(150);
    int3(5);
    resultBool = stl_inttest(&intchk);
    indicate_test_result (resultBool);


// test to check freq and ticks counter
// no interrupt on freq[2], ticks initial setting is 10
    for (i=0;i<9;i++)
    {
      int0(80);
      int1(15);
      int2(0);
      int3(5);
      resultBool = stl_inttest(&intchk);
      indicate_test_result (resultBool);

    }
   indicate_test_result (resultBool);        // no error yet, because ticks counter is decremented 9 times initilised with
10
   int0(80) ; int1(15) ; int2(0) ; int3(5) ;
   resultBool = stl_inttest(&intchk);
   indicate_test_result (resultBool);        // now error occures due to 10 times no interrupt in [2]
   int0(80) ; int1(15) ; int2(0) ; int3(5) ;
   resultBool = stl_inttest(&intchk);
   indicate_test_result (resultBool);        // now again error occures due to 11 times no interrupt in [2]
   int0(80) ; int1(15) ; int2(15) ; int3(5) ;
   resultBool = stl_inttest(&intchk);
   indicate_test_result (resultBool);        // now no error, because in [2] 15 interrupts occured

  }

}
```

```
/****************************************************/
/*                                                  */
/*                                                  */
/*  PROJECT      = 78K0 Selftest                    */
/*  DEVICE       = 78K0/KF2 (uPDF0547)              */
/*  TEST         = checksum over ROM area           */
/*                                                  */
/*               HL DE localy used -> on Stack      */
/*                                                  */
/*                                                  */
/****************************************************/


        NAME    stl_checksum          ; Label this program
        PUBLIC  stl_checksum          ;

        RSEG  CODE


stl_checksum:
        PUSH    HL                    ; Store HL on Stack
        PUSH    DE
        MOVW    HL, AX                ; Start Address into HL
        MOV     A, #0x00
next_rom:
        ADD     A,[HL]
        MOV     D, A                  ; buffer checksum in D
// check if all specified Memory is tested
        MOVW    AX, BC                ; move memory_test length indicator into AX
        SUBW    AX, #0x0001           ; decrement length indicator
        BZ      ROM_test_finished
        MOVW    BC, AX                ; buffer length indicator baxk to BC
        INCW    HL
        MOV     A, D                  ; restore checksum in A
        BR      next_rom
// Memory test finished, prepare return value
ROM_test_finished:
        MOV     A, D                  ; calculated cecksum into A ( return value )
        POP     DE
        POP     HL
        ret

        ENDMOD

        END
```

```
/*********************************************************/
/*                                                       */
/*  PROJECT      = 78K0 Selftest                         */
/*  DEVICE       = 78K0/KF2 (uPDF0547)                   */
/*  TEST         = Complement Test Routine               */
/*                                                       */
/*   in b/wwrite HL reg is used localy  -> Stack         */
/*   in b/wread  BC DE HL reg are used localy  -> Stack  */
/*                                                       */
/*                                                       */
/*********************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    COMP_test              ; Label this program
        PUBLIC  comp_bwrite
        PUBLIC  comp_wwrite
        PUBLIC  comp_bread
        PUBLIC  comp_wread
        PUBLIC  comp_detectederrors
        extern  comp_error

        RSEG  SADDR_Z(1)

comp_detectederrors     DS 2

wordflg                 DS8   1
#define Wordflag wordflg.0             // indicates in comp_xread if byte or word currently handeled
                                       // Wordflag =0 > byte  ///  Wordflag =1 > word
        RSEG  CODE

//*****************************************************************************
//      comp_byte_write
//
//      calling function -> comp_bwrite(&cb1,0x12) ;
//      B  contains the Data which have to be saved with complement (0x12)
//      AX contains the startaddress, were this data have to be saved (&cb1)
//      !!! it is recommended to disable interrupts during encoding process as proposed below
//
//*****************************************************************************
comp_bwrite:
        MOVW  HL, AX            // start address encoded data field into HL
        MOV   A, B              // store to be saved byte into A
//        DI                      // disable interrupt during write to preserve data intergrity
        MOV   [HL], A           // store original Byte into &cb1_data[0]
        XOR   A,  #11111111b    // generate complement of to be encoded data
        MOV   [HL + 1], A       // store complemented Byte into &cb1_data[1]
//        EI                      // enable interrupts again
        RET

//*****************************************************************************
//      comp_word_write
//
//      calling function -> comp_wwrite(&cw1,0x3456) ;
//      BC contains the data which have to be saved with complement (0x3456)
//      AX contains the startaddress, were this data have to be saved (&cw1)
//      !!! it is recommended to disable interrupts during encoding process as proposed below
//
//*****************************************************************************
comp_wwrite:
        MOVW  HL, AX            // start address encoded data field into HL
        MOVW  AX, BC            // store to be saved word into AX
//        DI                      // disable interrupt during write to preserve data intergrity
        MOV   [HL], A           // store original high byte into &cw1_data[0] high byte
        MOV   A, X              // mov original low byte into A
        MOV   [HL + 1], A       // store original low byte into &cw1_data[0] low byte
        XOR   A,  #11111111b    // generate complement of low byte of to be encoded data
        MOV   [HL + 3], A       // store complement of low byte into &cw1_data[1] low byte
        MOV   A, B              // mov original high byte into A
```

```
        XOR   A,  #11111111b    // generate complement of high byte of to be encoded data
        MOV   [HL + 2], A       // store complement of high byte into &cw1_data[1] high byte
//        EI                      // enable interrupts again
        RET

//*****************************************************************************
//      comp_byte_read
//
//      calling function -> cb = comp_bread(&cb1) ;
//      AX contains the startaddress, were this data have been stored ( &cb1 )
//      A  return value which will be stored in cb
//      !!! it is recommended to disable interrupts during decoding process as proposed below
//
//*****************************************************************************
comp_bread:
        clr1  S:Wordflag        // error source indication needed to give size to errorhandler
        MOVW  HL, AX            // start address of decoded data field into HL
//        DI                      // disable interrupt during write to preserve data intergrity
        MOV   A, [HL]           // load unencoded data into A
        MOV   X, A              // load unencoded data into X
        MOV   A, [HL + 1]       // load complemented data into A
//        EI                      // enable interrupts again
        XOR   A, #0xFF          // re-complementation
        SUB   A, X              // compare unencoded and complemented data
        BZ    cont1             // if zero skip jump to error_counter
        CALL  comp_error_asm    // if not zero jump to error_counter
cont1:  MOV   A, [HL]           // load unencoded data into A
        RET                     // return to cb = comp_bread(&cb1), A into cb

//*****************************************************************************
//      comp_word_read
//
//      calling function -> cw = comp_wread(&cw1) ;
//      AX  contains the startaddress, were this data have been stored ( &cw1 )
//      AX  return value which will be stored in cw
//      !!! it is recommended to disable interrupts during decoding process as proposed below
//
//*****************************************************************************
comp_wread:
        PUSH  BC
        set1  S:Wordflag        // error source indication needed to give size to errorhandler
        MOVW  HL, AX            // start address of decoded data field into HL
//        DI                      // disable interrupt during write to preserve data intergrity
        MOV   A, [HL]           // load unencoded data of high byte into A
        MOV   B, A              // buffer unencoded data of high byte into B for output
        MOV   X, A              // load unencoded data of high byte into X
        MOV   A, [HL + 2]       // load complemented data of high byte into A
        XOR   A, #0xFF          // re-complementation
        SUB   A, X              // compare unencoded and complemented data
        BZ    cont2             // if zero skip jump to error_counter
        CALL  comp_error_asm    // if not zero jump to error_counter
cont2:  MOV   A, [HL + 1]       // load unencoded data lower byte into A
        MOV   C, A              // buffer unencoded data of lower byte into C for return value
        MOV   X, A              // load unencoded data into X
        MOV   A, [HL + 3]       // load complemented data lower byte into A
//        EI                      // enable interrupts again
        XOR   A, #0xFF          // re-complementation
        SUB   A, X              // compare unencoded and complemented data
        BZ    cont3             // if zero skip jump to error_counter
        CALL  comp_error_asm    // if not zero jump to error_counter
cont3:  MOVW  AX, BC            // load original word into AX
        POP   BC
        RET                     // return to cw = comp_wread(&cw1), AX into cw

//*****************************************************************************
//      increment variable "comp_detectederrors" (16bit)
//
//      comp_detectederrors is incremented up to 0xFFFF and then it
//      remains at 0xFFFF, overflow is blocked
//*****************************************************************************
comp_error_asm:
        PUSH   BC
```

```
        PUSH    DE
        MOVW    AX, comp_detectederrors
        ADDW    AX, #0x0001       // increment value of "comp_detectederrors"
        BC      comp_error_counter_FF
                                  // check if overflow occured
        MOVW    comp_detectederrors, AX
                                  // store back incremented errorcounter

//*************************************************************************
//      prepare address, size and data of corrupted data to be used in comp_errorhandler
//
//      check if comp_errorhandler is activated
//      AX delivers *addr to error handler
//      B  delivers size ( byte = 1 or word = 2 ) to error handler
//      DE delivers unencoded part of corrupted data
//*************************************************************************
comp_error_counter_FF:
        MOVW  DE, #0x0000       // clear DE

        MOV   A, [HL]           // mov lower byte of corrupted data into A
        MOV   E, A              // and move it into E
        BTCLR S:Wordflag, sizechange  // check if error occured in B_read or W_read
        BR    comp_byte         // return --- now size is = 1 = byte
sizechange:
        MOV   B, #0x02          // B contains size parameter for comp_errorhandler routine 2=word
MOV   A, [HL + 1]       // in case of word move now upper byte of corrupted data into A
            MOV   D, A              // and move it into D
comp_byte:
        MOVW  AX, HL            // load starteddress of corrupted data into AX
        CALL  comp_error        // call comp_error_function
//in case of a detected error, return value( cb/cw ) will be defined/calculated by comp_error routine
        POP   DE
        POP   BC
        RET                     // return --- now size is = 2 = word

        ENDMOD
        END
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = Clock Test                        */
/*                                                   */
/*               AX, BC reg are used localy  -> Stack */
/*               A   return value                    */
/*                                                   */
/*****************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    stl_clocktest             ; Label this program
        PUBLIC  stl_clocktest

#define     mintime    0x3000       //lower limit of execution counter
#define     maxtime    0x3300       //upper limit of execution counter

        RSEG  SADDR_Z(1)

counter      DS     2   // operating clock counter
TMHMD1_AS    DS     1   // buffer during test Timer H1 Mode Register _ Application Settings
PM1_AS       DS     1   // buffer during test Port Mode Register 1 _ Application Settings
P1_AS        DS     1   // buffer during test Format of Port Register _ Application Settings
teststatus   DS     1   // to buffer teststatus


        RSEG  CODE
//*************************************************************************
//      stl_clocktest (example)
//      Internal low speed oscillator is generating together with Timer H1 a 20msec sqaure wave
//      signal at TOH1 pin. 20msec are usually available on most of Applications running on the mains.
//      This signal is then the inputsignal to port P15 ( pin 49 ). The number
//      instructions during the high periode ( 10msec ) are counted. This number gives a clear
//      indication whether the cpu is running on it's fundamental clock or on a harmonic.
//
//      This example is running with a main clock of 16.MHz. The number of instructions counted in the
//      16 cell "counter" is in this example approx. 0x313x. mintime is set to 0x3000, maxtime is 0x3300.
//      main clock of 8MHz results in a counter value of 0x18xx which would result in an error message.
//      !!!special attention in case of interrupts has to be taken by the user ( danger of wrong test_result )
//
//      A  returns 0x01 if a memory falt was detected, otherwise 0x00 is returned.
//*************************************************************************

stl_clocktest:
        PUSH    AX
        PUSH    BC

// prepare timer H1 to generate square wave signal at TOH1 ( pin 48 ) 20msec (50Hz)
        MOV     A, TMHMD1       // buffer during test register setting
        MOV     S:TMHMD1_AS, A  // buffer during test register setting
        MOV     A, PM1          // buffer during test port mode register settings
        MOV     S:PM1_AS, A     // buffer during test port mode register settings
        MOV     A, P1           // buffer during test format of port mode register settings
        MOV     S:P1_AS, A      // buffer during test format of port mode register settings

        clr1    PM1.6           // output mode for timer H1
        clr1    P1.6            // output mode

        set1    PM1.5           // input mode, 20msec input signal
        set1    P1.5            // input mode

        MOV     CMP01,  #00010010b // set to 18 to generate 20ms ( 50Hz ) outputsignal
        MOV     TMHMD1, #11010001b // timer enabled / fRL 1.8kHz / Interval timer / low level / output enabled
```

```
//synchronise onto low level
        MOVW    AX, #0x0000     // clear AX
wait_for_low:
        BT      P1.5, wait_for_low  // wait for low signal
wait_for_high:
        BF      P1.5, wait_for_high // if high signal, start counting
//        DI                        // disable interrupts during instruction counting periode
measure_high_time:
        INCW    AX
        BT      P1.5, measure_high_time // check port status
        MOVW    S:counter, AX   // if high periode is finished store counter value
//        EI                      // enable interrupts after counting periode

//check lower frequency limit
        CMPW    AX, #mintime
        BC      clock_error     // check lower limit

//check upper frequency limit
        MOVW    BC, AX          // count value into BC
        MOVW    AX, #maxtime    // maxtime into AX
        XCH     A, X
        SUB     A, C            // sbtract lower byte
        XCH     A, X
        SUBC    A, B            // subtract higher byte
        BC      clock_error
        MOV     A, #0x00
        MOV     teststatus, A   // teststatus TRUE, counter inside range
        BR      restore_regsettings

//set teststatus if error occured
clock_error:
        MOV     A, #0x01
        MOV     teststatus, A   // teststatus FALSE, counter out of range

restore_regsettings:
        MOV     A,S:TMHMD1_AS   // restore test register setting
        MOV     TMHMD1, A       // restore test register setting
        MOV     A, S:PM1_AS     // restore test port mode register settings
        MOV     PM1, A          // restore test port mode register settings
        MOV     A, S:P1_AS      // restore test format of port mode register settings
        MOV     P1, A           // restore test format of port mode register settings

        POP     BC
        POP     AX

        MOV     A, teststatus   // return value to calling function
        ret

        END
```

```
/*******************************************************/
/*                                                     */
/*                                                     */
/*  PROJECT      = 78K0 Selftest                       */
/*  DEVICE       = 78K0/KF2 (uPDF0547)                 */
/*  TEST         = Error correction code               */
/*                                                     */
/*   in b/wwrite DE HL reg are used localy  -> Stack   */
/*   in b/wread  BC DE HL reg are used localy  -> Stack*/
/*                                                     */
/*                                                     */
/*******************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    stl_ecctest            ; Label this program
        PUBLIC  ecc_bwrite
        PUBLIC  ecc_wwrite
        PUBLIC  ecc_bread
        PUBLIC  ecc_wread
        PUBLIC  ecc_detectederrors
        PUBLIC  ecc_fixederrors
        extern  ecc_error

        RSEG  SADDR_Z(1)

ecc_detectederrors          DS  2
ecc_fixederrors             DS  2

nibbelbuffer                DS8   1
nibbelcounter               DS8   1
Flags                       DS8   1
decodedA                    DS8   1

#define wordflag Flags.0
#define byteflag Flags.1

        RSEG  CONST
// ECC encoding table
ecc_encode:
        DB 0x42,0x4d,0x71,0x7e,0x17,0x18,0x24,0x2b,0xd4,0xdb,0xe7,0xe8,0x81,0x8e,0xb2,0xbd

// ECC decoding table
// The low order nibble of the following table is the decoded 4 bit value, but it is only valid if bit 7 is clear.
// Bit 7 is set if a 2 bit error has been encountered, which cannot be repaired. A 1 bit error can be fixed and it
// is indicated by bit 6 being set.
ecc_decode:
        DB 0x80,0x4c,0x40,0x80,0x46,0x81,0x80,0x44,0x45,0x81,0x80,0x47,0x81,0x41,0x4d,0x81
        DB 0x45,0x82,0x80,0x44,0x84,0x44,0x44,0x04,0x05,0x45,0x45,0x84,0x45,0x81,0x83,0x44
        DB 0x46,0x82,0x80,0x47,0x06,0x46,0x46,0x84,0x85,0x47,0x47,0x07,0x46,0x81,0x83,0x47
        DB 0x82,0x42,0x4e,0x82,0x46,0x82,0x83,0x44,0x45,0x82,0x83,0x47,0x83,0x4f,0x43,0x83
        DB 0x40,0x80,0x00,0x40,0x80,0x41,0x40,0x80,0x80,0x41,0x40,0x80,0x41,0x01,0x80,0x41
        DB 0x80,0x42,0x40,0x80,0x48,0x81,0x80,0x44,0x45,0x81,0x80,0x49,0x81,0x41,0x43,0x81
        DB 0x80,0x42,0x40,0x80,0x46,0x81,0x80,0x4a,0x4b,0x81,0x80,0x47,0x81,0x41,0x43,0x81
        DB 0x42,0x02,0x80,0x42,0x82,0x42,0x43,0x82,0x82,0x42,0x43,0x82,0x43,0x81,0x03,0x43
        DB 0x4c,0x0c,0x80,0x4c,0x86,0x4c,0x4d,0x84,0x85,0x4c,0x4d,0x87,0x4d,0x81,0x0d,0x4d
        DB 0x85,0x4c,0x4e,0x84,0x48,0x84,0x84,0x44,0x45,0x85,0x85,0x49,0x85,0x4f,0x4d,0x84
        DB 0x86,0x4c,0x4e,0x87,0x46,0x86,0x86,0x4a,0x4b,0x87,0x87,0x47,0x86,0x4f,0x4d,0x87
        DB 0x4e,0x82,0x0e,0x4e,0x86,0x4f,0x4e,0x84,0x85,0x4f,0x4e,0x87,0x4f,0x0f,0x83,0x4f
        DB 0x80,0x4c,0x40,0x80,0x48,0x81,0x80,0x4a,0x4b,0x81,0x80,0x49,0x81,0x41,0x4d,0x81
        DB 0x48,0x82,0x80,0x49,0x08,0x48,0x48,0x84,0x85,0x49,0x49,0x09,0x48,0x81,0x83,0x49
        DB 0x4b,0x82,0x80,0x4a,0x86,0x4a,0x4a,0x0a,0x0b,0x4b,0x4b,0x87,0x4b,0x81,0x83,0x4a
        DB 0x82,0x42,0x4e,0x82,0x48,0x82,0x83,0x4a,0x4b,0x82,0x83,0x49,0x83,0x4f,0x43,0x83

        RSEG  CODE
//*************************************************************************
//      ecc_byte/word_write
//
```

```
//      BC contains the Data which have to be encoded
//      AX contains the startaddress, were this data have to be saved
//*****************************************************************************
ecc_bwrite:
        PUSH  DE
        PUSH  HL
        MOVW  DE, AX            // start address encoded data field into DE
        clr1  S:wordflag        // byte write
        BR    ecc_encode_start

ecc_wwrite:
        PUSH  DE
        PUSH  HL
        MOVW  DE, AX            // start address encoded data field into DE
        set1  S:wordflag        // indicates word_write

ecc_encode_start:
//        DI                        //disable interrupts during encoding process if requiered
        MOV   A,  #0x00         // prepare clearing of decodeA
        MOV   decodedA, A       // clear decodeA, to give back correct bread value
// encoding upper nibbel
        MOVW  HL, #ecc_encode   // start address encoding table into DE
        MOVW  AX, BC            // mov to be encoded data into AX
loopw:  AND   A,  #0xF0         // isolate upper nibbel
        ROR   A,  1             // mov upper nibbel to lower nibbel location step 1
        ROR   A,  1             // step 2
        ROR   A,  1             // step 3
        ROR   A,  1             // step 4
        MOV   C,  A             // move isolated nibbel to C
        MOV   A, [HL+C]         // encoded upper nibbel value into A
        MOV   [DE], A           // encoded upper nibbel into &eb1/ew_data[0]
        INCW  DE                // DE contains no address of &eb1/ew_data[1]
// encoding lower nibbel
        MOV   A,  B             // move to be encoded value into A
        AND   A,  #0x0F         // isolate lower nibbel
        MOV   C,  A             // move lower nibbel into A
        MOV   A, [HL+C]         // encoded upper nibbel into A
        MOV   [DE], A           // encoded upper nibbel into &eb1_data[1]
        MOV   A, #0x00          // clear akku
        BF    S:wordflag, out   // check if word or byte encoding has to be done
        clr1  S:wordflag
        INCW  DE                // next address were encoded data have to be stored
        MOV   A,  X             // word encoding under progress, mov lower byte into A
        MOV   B,  A             // buffer lower byte into B
        BR    loopw             // continue Word encoding with lower byte
out:                            // byte/word encoding finished
        POP   HL
        POP   DE
//        EI                      // enable Interrupts again
        RET                     // return to main

//*****************************************************************************
//      ecc_byte/word_read
//
//      eb = ecc_bread(&eb1)     ew = ecc_wread(&ew1)
//      AX contains start address of encoded data (&eb1/&ew1)
//      AX returns decoded value to calling function into eb/ew
//*****************************************************************************
ecc_bread:
        PUSH  BC
        PUSH  DE
        PUSH  HL
        MOVW  DE, AX            // start address of to be decoded Byte
        set1  S:byteflag        // set byteflag - importend for correct byte return value
        MOV   A, #0x02          // 2 byte to be docoded
        MOV   nibbelcounter, A  // wordflag setting
        BR    ecc_decode_start  // start decoding

ecc_wread:
        PUSH  BC
        PUSH  DE
        PUSH  HL
```

```
        MOVW  DE, AX            // start address of to be decoded Word
        MOV   A, #0x04          // 4 byte to be docoded
        MOV   nibbelcounter, A  // wordflag setting

// start decoding check error indication bits ( bit 6 / bit 7 )
// bit 7 indicates 2bit error which is uncorrectable and results in jumping into ecc_error()
ecc_decode_start:
        MOVW  HL, #ecc_decode   // startaddress decoding table into HL
cont_decode:
        MOV   A, [DE]           // move 8bit encoded upper nibbel into A
        MOV   C,  A             // buffer encoded value into C
        MOV   A, [HL+C]         // catch decoded value from decode table
        ROL   A,  1             // mov bit 7 ( 2 bit error ) into CY
        BNC   tstbit6           // if no error occured skip error handling and check bit6
        MOV   S:nibbelbuffer,  A  // save A (decoded Value, under investigation) into S:nibbelbuffer
        MOVW  AX, ecc_detectederrors
        ADDW  AX, #0x0001       // increment ecc_detectederrors via A
        BC    ecc_de_counter_FF // tst if ecc_counter was 0xFFFF before incremented value stored
        MOVW  ecc_detectederrors, AX // store back incremented errorcounter
// prepare input data for ecc_error routine

// AX contains address of corrupted nibbel
// B indicates if byte or word was encoded
// D contains the corrupted nibbel
        MOV   B, #0x01          // indicates ecc_error routine byte is encoded
        BT    S:byteflag, size_byte // check if byte is encoded
        MOV   B, #0x02          // if byteflag is not set, word is encoded
size_byte:
        MOVW  AX, DE            // address of encoded corrupted data
        MOVW  HL, AX            // and buffer this in HL
        MOV   A,[DE]            // reload corrupted data into A
        PUSH  DE                // buffer address [DE]of encoded data field onto stack
        MOV   D, A              // corrupted data now in D
        MOVW  AX, HL            // address of corrupted data now in AX

        CALL  ecc_error         // jump into ecc_error(), define there what to do in case of 2bit error

        POP   DE                // restore buffer address [DE]of encoded data field from stack
        MOV   A,  S:nibbelbuffer//  bring back decoded Byte under investigation into A
        BR    str_nibbel        //  continue ecc decoding in ecc_xread function
                                // after returning from ecc error routine leave ecc_read function
                                // returnvalue (AX) in case of detected unrepairable error must be defined in
        POP   BC                // ecc_error routine
        POP   DE
        POP   HL
        RET
  // to be inserted later ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ceck if errorhandler is activated if yes prepare address of corrupted data and size
  // NO_ECC - ECC_NOWRITEBACK not implemented yet
ecc_de_counter_FF:
        MOV   A,  S:nibbelbuffer//  bring back decoded Byte under investigation into A
tstbit6:
        ROL   A,  1             // mov bit 6 ( repaired error ) into CY
        BNC   str_nibbel        // if no error indicated by bit 6 continue , store nibbel
        MOV   S:nibbelbuffer, A // save A (decoded Value, under investigation)  into nibbelbuffer
        MOVW  AX, ecc_fixederrors
        ADDW  AX, #0x0001
        BC    ecc_fix_counter_FF// tst if ecc_counter was 0xFF before INC
        MOVW  ecc_fixederrors, AX // store back incremented errorcounter
ecc_fix_counter_FF:
        MOV   A,  S:nibbelbuffer//  bring back decoded Byte under investigation into A

  // to be inserted later ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // ceck if errorhandler is activated if yes prepare address of corrupted data and size
  // NO_ECC - ECC_NOWRITEBACK not implemented yet

// extract nibbel by nibbel
str_nibbel:
// check which nibbel has to be decoded now
        MOV   X,  A             // buffer to be decoded nibbel which is already two times roled left in X
```

```
        MOV   A,  #0x04         // compare value to nibbelcounter
        CMP   A,  nibbelcounter // compare nibbelcounter with 0x04
        BNZ   tst3              // if not 0x04 check if it is 0x03
// decoding first nibbel of a word is under progress
        MOV   A,  X             // reload lower nibbel from X
        ROL   A,  1             // align decoded nibbel into upper nibbel
        ROL   A,  1             // align decoded nibbel into upper nibbel
        AND   A,  #11110000b    // lower nibbel put to zero
        MOV   B,  A             // buffer upper nibbel of 1st word byte into B
        MOV   A,  #0x03         // prepare decrementation of nibbelcounter
        MOV   nibbelcounter, A  // decrement nibbelcounter
        BR    cont              // continue decoding process
// check which nibbel has to be decoded now
tst3:
        MOV   A,  #0x03         // comparevalue for nibbelcounter
        CMP   A,  nibbelcounter // check if nibbel three has to be decoded
        BNZ   tst2              // if not check if nibbel 2 has to be decoded now
// decoding second nibbel of a word is under progress
        MOV   A,  X             // reload lower nibbel from X
        ROR   A,  1             // align decoded nibbel into lower nibbel
        ROR   A,  1             // align decoded nibbel into lower nibbel
        AND   A,  #00001111b    // upper nibbel put to zero
        OR    A,  B             // put together A ( lower nibbel ) and B ( upper nibbel )
        MOV   decodedA,  A      // store upper word byte in "decodedA"
        MOV   A,  #0x02         // prepare decrementation of nibbelcounter
        MOV   nibbelcounter, A  // decrement nibbelcounter
        BR    cont              // continue decoding process
// check which nibbel has to be decoded now
tst2:
        MOV   A,  #0x02         // comparevalue for nibbelcounter
        CMP   A,  nibbelcounter // check if nibbel two has to be decoded
        BNZ   tst1              // if not nibbel one has to be decoded
// decoding third nibbel of a word or first nibbel of a byte is under progress
        MOV   A,  X             // reload lower nibbel from X
        ROL   A,  1             // align decoded nibbel into upper nibbel
        ROL   A,  1             // align decoded nibbel into upper nibbel
        AND   A,  #11110000b    // lower nibbel forced to zero
        MOV   B,  A             // buffer upper nibbel of 2nd word byte or to be decoded byte into B
        MOV   A,  #0x01         // prepare decrementation of nibbelcounter
        MOV   nibbelcounter, A  // decrement nibbelcounter
        BR    cont              // continue decoding process
// decode last nibbel
tst1:
        MOV   A,  X             // reload lower nibbel from X
        ROR   A,  1             // align decoded nibbel into lower nibbel
        ROR   A,  1             // align decoded nibbel into lower nibbel
        AND   A,  #00001111b    // upper nibbel forced to zero
        OR    A,  B             // put together A ( lower nibbel ) and B ( upper nibbel )
        BT    S:byteflag, byteout
        MOV   X,  A             // decoded byte in X or lower byte of decoded word in X
        MOV   A,  decodedA      // move upper byte of decoded word into A ( in case of byte decoding decodedA = 0 )
        POP   BC
        POP   DE
        POP   HL
        RET                     // return to main from wordread
byteout:
        clr1  S:byteflag        // char eb will be filled by 8bit X register !!!
        POP   BC
        POP   DE
        POP   HL
        RET                     // return to main from byteread

cont:   INCW  DE                // next address of to be decoded byte
        BR  cont_decode         // continue decoding process with next byte

        ENDMOD

        END
```

```
//*****************************************************************************
//
//      Self Test Library
//
//*****************************************************************************
//      stl_ecc_comp_error_routines
//
//      In case of a detected error in stl_ess_test.asm or stl_comp_test.asm
//      the functions comp_error resp. ecc_error will be executed.
//      This functions are empty yet. In dependency on the final application,
//      customer has to define here actions to be undertaken in case of detected errors
//
//*****************************************************************************


#include <intrinsics.h>
//#include <io78f0547_80.h>

//extern unsigned int comp_errorhandler;
//extern unsigned int ecc_errorhandler;
//extern unsigned int comp_error;
//extern unsigned int ecc_error;

//void stl_myinit(void)
//{
//   stl_init() ; /* do the library initialization */
//   stl_control = NO_ECC|ECC_NOWRITEBACK ;     /* do not repair correctable errors; do not write back corrected data */
//   stl_control = ECC_NOWRITEBACK ;            /* do not write back corrected data */
//   stl_control = 2;                           /* do repair correctable errors, do not write back corrected data*/
//   comp_errorhandler = &comp_error ;
//   ecc_errorhandler = &ecc_error ;
//}

volatile unsigned int *d1;   // just for test purpose
volatile unsigned char d2;   // just for test purpose
volatile unsigned int d3;    // just for test purpose

// application dependent comp error correction routine
unsigned int comp_error(void *addr, unsigned char size, unsigned int data)
{
   __no_operation();
   d1=addr;                 // just for test purpose // startaddress of unencoded error data
   d2=size;                 // just for test purpose // 0x01 is byte error 0x02 is word error
   d3=data;                 // just for test purpose // unencoded part of corrupted data
   return(data);
}

// application dependent ecc error correction routine
// this routine will be executed if an 2-bit error has been detected in ecc_xread
unsigned int ecc_error(void *addr, unsigned char size, unsigned char data)
{
   __no_operation();
   d1=addr;                 // just for test purpose // startaddress of error data
   d2=size;                 // just for test purpose // 0x01 is byte error 0x02 is word error
   d3=data;                 // just for test purpose // unencoded part of corrupted data
   return(data) ;
}
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = Interrupt Test                    */
/*                                                   */
/*  BC, DE, HL  reg are used locally  -> Stack       */
/*                                                   */
/*                                                   */
/*****************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>
#include <stl.h>

        NAME    stl_inttest                    // Label this program
        PUBLIC  stl_inttest

        RSEG  SADDR_Z(1)

address_0_freq          DS  2                  // address of freq[0]
address_0_ticks         DS  2                  // address of ticks[0]
address_0_freq_lower    DS  2                  // address of freq_lower
address_0_freq_upper    DS  2                  // address of freq_upper
address_0_freq_initial  DS  2                  // address of freq_initial
address_0_ticks_initial DS  2                  // address of ticks_initial

array_start_address     DS  2                  // startaddress of array ( struct )
buffer_freq             DS  2                  // buffer counted interrupts
buffer_calculated_freq  DS  2                  // result of ( freq_init - freq )
buffer_initial_freq     DS  2                  // buffer initial frequency
buffer_initial_ticks    DS  2                  // buffer initial ticks
array_size              DS  1                  // number of Interrupts to be monitored ( 127 max )

Flags                   DS8 1                  // 1 byte flags
#define error_flag Flags.0                     // error_flag

GET_WRD MACRO                                  // Get Word specified by [AX]->[HL] into AX
        MOVW     HL, AX                        // address of freq_xxx[0] now into HL
        MOV      A, [HL + C]                   // add array offset onto freq_xxx[0]
        MOV      X, A                          // lower byte of freq_xxx [X] into X
        INC      C                             // now address upper byte of freq_xxx [X]
        MOV      A, [HL + C]                   // add array offset onto freq_xxx[0]
        DEC      C                             // now freq_xxx [X] is in AX
        ENDM

SUB_WRD MACRO                                  // SUB DE from AX, result in AX ( AX = AX - DE )
        XCH     A, X                           // exchange A and X to start substraction with low byte
        SUB     A, E                           // substract low byte
        XCH     A, X                           // exchange A and X again
        SUBC    A, D                           // substract high bytes, result now in AX
        ENDM

WRT_WRD MACRO                                  // Write Word ( AX ) into Byte specified by [ HL ]
        XCH     A, X                           // exchange A and X
        MOV     [HL + C], A                    // move low byte into low byte  of [HL]
        XCH     A, X                           // excange A and X
        INC     C                              // increment address offset
        MOV     [HL + C], A                    // move high byte of high byte of [HL]
        DEC     C                              // readjust address offset
        ENDM


        RSEG  CODE
//*****************************************************************************
//      STL_Interrupt test
//      array_size contains number of Interrupts to be monitored. max 127
//      during test routine Interrupts should be disabled, because this can influence
//      the test result
//
```

```
//
//      A  returns 0x01 if a memory falt was detected, otherwise 0x00 is returned.
//*************************************************************************

stl_inttest:

        PUSH    BC                              // buffer
        PUSH    DE                              // register
        PUSH    HL                              // content

        CLR1    S:error_flag                    // clear error flag
//        DI                                      // disable interrupts if requiered

//buffer array start address
        MOVW    S:array_start_address, AX       // buffer struct start address
        MOVW    HL, AX                          // array start address into HL
        MOVW    AX, #0x0000                     // clear AX
        MOVW    BC, AX                          // clear BC
//buffer array size into "array_size"
        MOV     A, [HL + INTCHK_ARRAY_SIZE]     // move array size into A
        MOV     S:array_size, A                 // store array size
        DEC     A
        ROL     A, 1                            // multiplication with 2 ( word adaption )
        MOV     C, A                            // C carries loop counter for array
//generate total address field
        MOV     A, [HL]                         // lower part of array start address into A
        MOV     X, A                            // lower part of array startaddress into lower part of AX
        INCW    HL                              // inc HL to catch upper part of array startaddress
        MOV     A, [HL]                         // upper part of array start address into A
        DECW    HL                              // dec HL , array startaddress
        MOVW    S:address_0_freq, AX            // address of freq[0] into RAM address_0_freq
// address ticks[0]
        MOV     A, [HL + INTCHK_TICKS]
        MOV     X, A
        INCW    HL
        MOV     A, [HL + INTCHK_TICKS]
        DECW    HL
        MOVW    S:address_0_ticks, AX           // address of ticks[0] into RAM address_0_ticks
// address freq lower[0]
        MOV     A, [HL + INTCHK_FREQ_LOWER]
        MOV     X, A
        INCW    HL
        MOV     A, [HL + INTCHK_FREQ_LOWER]
        DECW    HL
        MOVW    S:address_0_freq_lower, AX      // address of freq_lower[0] into RAM address_0_freq_lower
// address freq upper[0]
        MOV     A, [HL + INTCHK_FREQ_UPPER]
        MOV     X, A
        INCW    HL
        MOV     A, [HL + INTCHK_FREQ_UPPER]
        DECW    HL
        MOVW    S:address_0_freq_upper, AX      // address of freq_upper[0] into RAM address_0_freq_upper
// address freq initial[0]
        MOV     A, [HL + INTCHK_FREQ_INITIAL]
        MOV     X, A
        INCW    HL
        MOV     A, [HL + INTCHK_FREQ_INITIAL]
        DECW    HL
        MOVW    S:address_0_freq_initial, AX    // address of freq_initial[0] into RAM address_0_freq_initial
// address freq ticks[0]
        MOV     A, [HL + INTCHK_TICKS_INITIAL]
        MOV     X, A
        INCW    HL
        MOV     A, [HL + INTCHK_TICKS_INITIAL]
        DECW    HL
        MOVW    S:address_0_ticks_initial, AX   // address of ticks_initial[0] into RAM address_0_ticks_initial
```

```
// start checking
next_interrupt_source:

// load ticks_initial into buffer_initial_ticks
        MOVW     AX, S:address_0_ticks_initial  // address of freq_initial[0] into AX
        GET_WRD                                 // now freq_initial[X] is in AX
        MOVW    S:buffer_initial_ticks, AX      // buffer initial frequency for reinitilisation in next step

//calculate f[x] = freq_init[x] - freq[x]
// 1. load freq into DE
        MOVW     AX, S:address_0_freq           // address of freq[0] into AX
        GET_WRD                                 // Get Word specified by [AX]->[HL] into AX
        MOVW     DE, AX                         // now freq[X] is in DE
        MOVW     S:buffer_freq, AX              // buffer counted intrrupts
// 2. load freq_initial into AX
        MOVW     AX, S:address_0_freq_initial   // address of freq_initial[0] into AX
        GET_WRD                                 // now freq_initial[X] is in AX
        MOVW    S:buffer_initial_freq, AX       // buffer initial frequency for reinitilisation in next step
// 3. calculate f[x], AX - DE = f in AX
        SUB_WRD                                 // SUB DE from AX, result in AX ( AX = AX - DE )
        MOVW    S:buffer_calculated_freq, AX    // result now in calculated_f

// reinitilise freq with freq_init
        MOVW    AX, S:address_0_freq            // address of freq[0] into AX
        MOVW    HL, AX                          // and now into HL
        MOVW    AX, S:buffer_initial_freq       // move initial frequency into AX
        WRT_WRD                                 // Write Word ( AX ) into Byte specified by [ HL ]

// check if calculated freq is within the limit of freq_lower and freq_upper
// 1. freq_calculated - freq_lower then check C flag if C flag 1 then set error
        MOVW    AX, S:address_0_freq_lower      // address of freq_lower[0] into AX
        GET_WRD                                 // Get Word specified by [AX]->[HL] into AX
        MOVW    DE, AX                          // freq_lower limit now in DE
        MOVW    AX, S:buffer_calculated_freq
        SUB_WRD
        BNC     OK_Lower_limit
        SET1    S:error_flag                    // set error flag
// 2. freq_upper - freq_calculated then check C flag. if C flag 1 then set error
OK_Lower_limit:
        MOVW    AX, S:buffer_calculated_freq
        MOVW    DE, AX
        MOVW    AX, S:address_0_freq_upper      // address of freq_upper[0] into AX
        GET_WRD                                 // Get Word specified by [AX]->[HL] into AX
        SUB_WRD                                 // SUB DE from AX, result in AX ( AX = AX - DE )
        BNC     OK_Upper_limit
        SET1    S:error_flag                    // set error flag

// check if any interrupt occured
// if buffer_freq = 0, no interrupt occured
OK_Upper_limit:
        MOVW    AX, S:buffer_calculated_freq    // move number of counted interrupts into AX
        CMPW    AX, #0x0000                     // check if number of counted interrupts is 0
        BNZ     init_ticks                      // interrupt occured now reinitialize ticks counter
// no interrupt occured -> decrement ticks ( not below zero )
        MOVW    AX, S:address_0_ticks           // address of ticks[0] into AX
        GET_WRD                                 // now content of ticks[C] is in AX
        CMPW    AX, #0x0000                     // check if ticks[C] is already zero
        BZ      skip_decw_ax                    // if already zero no further decrement
        DECW    AX                              // decrement tick counter
        WRT_WRD                                 // write back tick_counter_value into ticks[C]

// ticks == 0 ?
skip_decw_ax:
        CMPW    AX, #0x0000                     // check again after last decrement if ticks[C] is zero
        BNZ     dec_array_size                  // if not zero, decremnt array_size ( loop counter )

// ticks_init is 0 ?
check_tick_init:
        MOVW    AX, S:buffer_initial_ticks      // initial_ticks[C] into AX
        CMPW    AX, #0x0000                     // check if initial_ticks[C] is zero
```

```
        BZ      dec_array_size                  // initial_ticks[C] is zero, then jump to dec loop counter

// ticks_init not zero
        SET1    S:error_flag                    // set error flag
        BR      dec_array_size                  // next loop

// reinitilise ticks with ticks_init
init_ticks:
        MOVW    AX, S:address_0_ticks           // address of freq[0] into AX
        MOVW    HL, AX                          // and now into HL
        MOVW    AX, S:buffer_initial_ticks      // move initial frequency into AX
        WRT_WRD                                 // Write Word ( AX ) into Byte specified by [ HL ]

// decrement array_size ( loop_counter ) if not C = 1
dec_array_size:
        DEC     C
        DEC     C                               // two times decrement due to word processing ( two byte )
        MOV     A, C
        CMP     A, #0xFE                        // check if loop counter
        BZ      out                             // if zero flag set test finished
        BR      next_interrupt_source           // next array element

out:    POP     HL                              // restore
        POP     DE                              // buffer
        POP     BC                              // content
// prepare return value ( boolean )
        BT      S:error_flag, error_occured     // check error flag
        MOV     A, #0x00                        // return value ( boolean ) in A
//        EI
        ret
error_occured:
        MOV     A, #0x01                        // return value ( boolean ) in A
//        EI
        ret

        END
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = Program Counter Test - Signatur    */
/*                                                   */
/*                                                   */
/*****************************************************/

// Adressing via Linker - XCL File - Signatur

// Standard ICC78K include files


#include <io78f0547_80.h>

        NAME    PC_Test_signature        ; Label this program
        PUBLIC  stl_pctest               ;

        RSEG    CODE

//*************************************************************************
//      stl program counter test
//
//      extra XCL file defines via Linker fixed code segments, where the Programm
//      counter jumps to
//
//*************************************************************************

stl_pctest:
        CALL    PCTEST1                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST1             // compare AX, loaded in subr PCTEST1 with address of PCTEST1
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST2                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST2             // compare AX, loaded in subr PCTEST2 with address of PCTEST2
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST3                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST3             // compare AX, loaded in subr PCTEST3 with address of PCTEST3
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST4                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST4             // compare AX, loaded in subr PCTEST4 with address of PCTEST4
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST5                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST5             // compare AX, loaded in subr PCTEST5 with address of PCTEST5
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST6                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST6             // compare AX, loaded in subr PCTEST6 with address of PCTEST6
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST7                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST7             // compare AX, loaded in subr PCTEST7 with address of PCTEST7
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST8                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST8             // compare AX, loaded in subr PCTEST8 with address of PCTEST8
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTEST9                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTEST9             // compare AX, loaded in subr PCTEST9 with address of PCTEST9
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTESTA                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
        CMPW    AX, #PCTESTA             // compare AX, loaded in subr PCTESTA with address of PCTESTA
        BNZ     PC_Error                 // compare - should be ZERO - otherwise an error occured - leave pc_test
        CALL    PCTESTB                  // PC jumps to ROM addr, predefined by file "stl_pc_const.xcl"
```

```
        CMPW    AX, #PCTESTB            // compare AX, loaded in subr PCTESTB with address of PCTESTB
        BNZ     PC_Error                // compare - should be ZERO - otherwise an error occured - leave pc_test

        MOV     A,  #0x00               // stl_pctest ok without errors, return value 0x00 returned to
        ret                             // calling function via register A

PC_Error:
        MOV     A,  #0x01               // stl_pctest detected an err., return value 0x01 returned to
        ret                             // calling function via register A

        RSEG    PCTEST_1                // start of relocatable segment PCTEST_1
PCTEST1:
        MOVW    AX, #SFB(PCTEST_1)      // SFB means Segment Begin - Startaddr des Seg PCTEST_1 in AX
        RET
        RSEG    PCTEST_2                // start of relocatable segment PCTEST_2
PCTEST2:
        MOVW    AX, #SFB(PCTEST_2)      // SFB means Segment Begin - Startaddr des Seg PCTEST_2 in AX
        RET
        RSEG    PCTEST_3                // start of relocatable segment PCTEST_3
PCTEST3:
        MOVW    AX, #SFB(PCTEST_3)      // SFB means Segment Begin - Startaddr des Seg PCTEST_3 in AX
        RET
        RSEG    PCTEST_4                // start of relocatable segment PCTEST_4
PCTEST4:
        MOVW    AX, #SFB(PCTEST_4)      // SFB means Segment Begin - Startaddr des Seg PCTEST_4 in AX
        RET
        RSEG    PCTEST_5                // start of relocatable segment PCTEST_5
PCTEST5:
        MOVW    AX, #SFB(PCTEST_5)      // SFB means Segment Begin - Startaddr des Seg PCTEST_5 in AX
        RET
        RSEG    PCTEST_6                // start of relocatable segment PCTEST_6
PCTEST6:
        MOVW    AX, #SFB(PCTEST_6)      // SFB means Segment Begin - Startaddr des Seg PCTEST_6 in AX
        RET
        RSEG    PCTEST_7                // start of relocatable segment PCTEST_7
PCTEST7:
        MOVW    AX, #SFB(PCTEST_7)      // SFB means Segment Begin - Startaddr des Seg PCTEST_7 in AX
        RET
        RSEG    PCTEST_8                // start of relocatable segment PCTEST_8
PCTEST8:
        MOVW    AX, #SFB(PCTEST_8)      // SFB means Segment Begin - Startaddr des Seg PCTEST_8 in AX)
        RET
        RSEG    PCTEST_9                // start of relocatable segment PCTEST_9
PCTEST9:
        MOVW    AX, #SFB(PCTEST_9)      // SFB means Segment Begin - Startaddr des Seg PCTEST_9 in AX
        RET
        RSEG    PCTEST_A                // start of relocatable segment PCTEST_A
PCTESTA:
        MOVW    AX, #SFB(PCTEST_A)      // SFB means Segment Begin - Startaddr des Seg PCTEST_A in AX
        RET
        RSEG    PCTEST_B                // start of relocatable segment PCTEST_B
PCTESTB:
        MOVW    AX, #SFB(PCTEST_B)      // SFB means Segment Begin - Startaddr des Seg PCTEST_B in AX
        RET

        ENDMOD

        END
```

```
//-------------------------------------------------------------------------
//      XLINK command file template for 78K0 microcontroller uPD78F0547_80.
//
//      This file is used in conjunction with stl_pc_test and defines the
//      Program counter jumps onto the defined addresses, to test it's correct
//      funktion
//
//      As ROM area is different from device to device, the addresses where
//      the PC should jump to have to be defined by the user in his final
//      application
//
//      the uPD78F0547_80, where this example is written for has a ROM area
//      from 0x0000 to 0xBFFF, there fore the PC jump addresses have been
//      defined as below
//
//-------------------------------------------------------------------------


-Z(CODE)PCTEST_B=BBBB
-Z(CODE)PCTEST_A=AAAA
-Z(CODE)PCTEST_9=9999
-Z(CODE)PCTEST_8=8888
-Z(CODE)PCTEST_7=7777
-Z(CODE)PCTEST_6=6666
-Z(CODE)PCTEST_5=5555
-Z(CODE)PCTEST_4=4444
-Z(CODE)PCTEST_3=3333
-Z(CODE)PCTEST_2=2222
-Z(CODE)PCTEST_1=1111
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = RAM checker patter Test           */
/*                                                   */
/*              DE, HL onto Stack, used localy       */
/*                                                   */
/*                                                   */
/*****************************************************/
// Standard ICC78K include files


#include <io78f0547_80.h>

        NAME    stl_ram_cb_testb      ; Label this program
        PUBLIC  stl_ram_cb_testb      ;


        RSEG  SADDR_Z(1)
errorindicator:   DS   1
#define errorflg errorindicator.0

        RSEG  CODE

//*************************************************************************
//      stl_ram_cb_testb
//
//      this is an no destructive RAM test
//      calling function -> resultBool = stl_ram_cb_testb( addr, num );
//      AX  contains the startaddress, of the RAM under test ( addr )
//      BC  contains the number of bytes to be tested ( must be an even number,
//      because RAM location n and n+1 is done in one test step
//
//      !!! in dependency of the user application it might be necessary to
//      dissable/enable interrupts while testing, original RAM content is
//      buffered during test in CPU register
//      in this routine interrupts are dissabled/ enabled in short time intervals
//      ( the whole test in done in 2Byte units, interrupt is disabled and enabled
//      for each of the 2Byte units )
//
//      AX  return  a bool value to indicate an error ( 01 ) or not ( 00 )
//*************************************************************************
stl_ram_cb_testb:
        PUSH    DE                    ; Store DE on Stack
        PUSH    HL                    ; Store HL on Stack
        MOVW    HL, AX                ; Start Address into HL
        clr1    S:errorflg
next_ram:
//        DI                            ; disable interrupts for the time 2 bytes are tested
// buffer RAM and RAM+1 under test in DE
        MOV     A, [HL+1]             ; [HL] addressed RAM cell into A
        MOV     E, A                  ; Buffer RAM cell in E during test
        MOV     A, [HL]
        MOV     D, A                  ; now RAM and RAM+1 under test is buffered in DE
// write testpattern 0x55 / 0xAA into RAM n and n+1
        MOV     A, #0x55              ; write testpattern 0x55 into A
        MOV     [HL], A               ; write testpattern 0x55 into [HL] addressed memory
        MOV     A, #0xAA              ; write testpattern 0xAA into A
        MOV     [HL+1], A             ; write testpattern 0xAA into [HL+1]
// verify testpattern 0x55 / 0xAA
        MOV     A, #0x00              ; clear A
        MOV     A, [HL]               ; read back 0x55 testpattern
        CMP     A, #0x55              ; compare from RAM under test read out testpattern with reference
        BZ      no_error_0x55
        SET1    S:errorflg
no_error_0x55:
        MOV     A, [HL+1]             ; read back 0xAA testpattern
        CMP     A, #0xAA              ; compare from RAM under test read out testpattern with reference
        BZ      no_error_0xAA
        SET1    S:errorflg
```

```
no_error_0xAA:
// write testpattern 0xAA / 0x55 into RAM n and n+1
        MOV     A, #0xAA              ; write testpattern 0xAA into A
        MOV     [HL], A               ; write testpattern 0xAA into [HL] addressed memory
        MOV     A, #0x55              ; write testpattern 0x55 into A
        MOV     [HL+1], A             ; write testpattern 0x55 into [HL+1]
// verify testpattern 0xAA / 0x55
        MOV     A, #0x00              ; clear A
        MOV     A, [HL]               ; read back 0xAA testpattern
        CMP     A, #0xAA              ; compare from RAM under test read out testpattern with reference
        BZ      no_error_0xAA_inv
        SET1    S:errorflg
no_error_0xAA_inv:
        MOV     A, [HL+1]             ; read back 0x55 testpattern
        CMP     A, #0x55              ; compare from RAM under test read out testpattern with reference
        BZ      no_error_0x55_inv
        SET1    S:errorflg
no_error_0x55_inv:
// restore RAM and RAM+1 under test
        MOV     A, E
        MOV     [HL+1], A
        MOV     A, D
        MOV     [HL], A
//        EI                           ; enable interrupts again
// check if all specified RAM is tested
        MOVW    AX, BC                ; move RAM_test length indicator into AX
        SUBW    AX, #0x0002           ; decrement length indicator by 2 ( testing is done in pairs if memory )
        BZ      RAM_test_finished
        MOVW    BC, AX                ; buffer length indicator baxk to BC
        INCW    HL                    ; HL + 2, RAM testing is done pairwise
        INCW    HL
        BR      next_ram
// RAM test finished, prepare return value
RAM_test_finished:
        POP     HL
        POP     DE
        BT      S:errorflg, out_error
        MOV     A, #0x00              // give back 00 ( no error ) to calling function

        ret
out_error:
        MOV     A, #0x01              // give back 01 ( error )to calling function
        ret



        END
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = March -C RAM Test                 */
/*                                                   */
/*               HL DE reg are used localy  -> Stack  */
/*                                                   */
/*                                                   */
/*****************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    stl_ram_marchcmb            ; Label this program
        PUBLIC  stl_ram_marchcmb

        RSEG  SADDR_Z(1)
marchc_lgth    DS   2
marchc_start   DS   2
marchc_stop    DS   2

        RSEG  CODE
//*************************************************************************
//      stl_ram_marchcmb
//
//      AX contains the start address of to be tested RAM area
//      BC contains the length of the to be tested RAM area
//
//      A  returns 0xFF if a memory falt was detected, otherwise 0x00 is returned.
//*************************************************************************

stl_ram_marchcmb:
        PUSH DE
        PUSH HL                 // store HL register
        MOVW  S:marchc_start, AX// store start address of marchc test
        MOVW  HL, AX            // start address into HL
        MOVW  AX, BC            // test length into AX
        MOVW  S:marchc_lgth, AX // store test length
// test if length is zero
        SUBW  AX, #0x0001       // decrement test_length_value
        BNC   length_not_zero   // if test_length_value was zero, c flag is set
        MOV   A, #0x00          // return value true - no error
        POP  HL
        POP  DE
        ret
// start march -c test
length_not_zero:
        MOVW  AX, S:marchc_lgth // reload test length into AX
        XCH   A, X              // calculate march-c stop address
        ADD   A, L              // add lower byte
        XCH   A, X
        ADDC  A, H              // add higher byte
        DECW  AX                // adjust stop value ( n-1 )
        MOVW  S:marchc_stop, AX // store march-c end address
// step 1 - <>(w0) - clear specified memory area
        MOV   A, #0x00          // clear register A
        MOV   D, A              // clear D, which functions as error indicator
loop_step_1:
        MOV   [HL], A           // clear [HL] content
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_2    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_1       // next cycle

// step 2 - >(r0,w1) - read bitweise if zero is in RAM, then write 1 into it, from lower address
march_c_step_2:
```

```
        MOVW  AX, S:marchc_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchc_start// start address into AX
        MOVW  HL, AX            // start address into HL
loop_step_2:
        CALL  r0_w1             // read 0, write 1 to by HL addressed byte
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_3    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_2       // next cycle

// step 3 - >(r1,w0) - read bitweise if one is in RAM, then write zero into it, from lower address
march_c_step_3:
        MOVW  AX, S:marchc_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchc_start// start address into AX
        MOVW  HL, AX            // start address into HL
loop_step_3:
        CALL  r1_w0             // read 1, write 0 to by HL addressed byte
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_4    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_3       // next cycle

// step 4 - <(r0,w1) - read bitweise if zero is in RAM, then write one into it, from upper addess
march_c_step_4:
        MOVW  AX, S:marchc_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchc_stop // stop address into AX
        MOVW  HL, AX            // stop address into HL
loop_step_4:
        CALL  r0_w1             // read 0, write 1 to by HL addressed byte
        DECW  HL                // decrement RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_5    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_4       // next cycle

// step 5 - <(r1,w0) - read bitweise if one is in RAM, then write zero into it from upper address
march_c_step_5:
        MOVW  AX, S:marchc_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchc_stop // stop address into AX
        MOVW  HL, AX            // stop address into HL
loop_step_5:
        CALL  r1_w0             // read 1, write 0 to by HL addressed byte
        DECW  HL                // decrement RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_6    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_5       // next cycle

// step 6 - <>(r0) - check if specified memory area is all zero
march_c_step_6:
        MOVW  AX, S:marchc_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchc_start// start address into AX
        MOVW  HL, AX            // start address into HL
loop_step_6:
        MOV   A, [HL]           // load RAM cell addressed by HL
        CMP   A, #0x00          // check if RAM content is zero
        BZ    is_zero
        MOV   D, #0x01          // D functions as error indicator
is_zero:
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
```

```
        SUBW  AX, #0x0001       // decrement length counter
        BZ    out_6             // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_6       // next cycle

// move test result into reg A and return it to calling function
out_6:
        MOV   A, D              // return test result
        POP   HL                // restore HL content
        POP   DE                // restore DE content
        ret

// subroutine to read by HL addressed ram, check bitwise if content is 0 and set bitweise to 1
r0_w1:
        MOV   A, [HL]           // move to be tested byte into reg A
        CMP   A, #00000000b
        BZ    S_HL_0
        MOV   D, #0x01
S_HL_0: set1  [HL].0
        MOV   A, [HL]
        CMP   A, #00000001b
        BZ    S_HL_1
        MOV   D, #0x01
S_HL_1: set1  [HL].1
        MOV   A, [HL]
        CMP   A, #00000011b
        BZ    S_HL_2
        MOV   D, #0x01
S_HL_2: set1  [HL].2
        MOV   A, [HL]
        CMP   A, #00000111b
        BZ    S_HL_3
        MOV   D, #0x01
S_HL_3: set1  [HL].3
        MOV   A, [HL]
        CMP   A, #00001111b
        BZ    S_HL_4
        MOV   D, #0x01
S_HL_4: set1  [HL].4
        MOV   A, [HL]
        CMP   A, #00011111b
        BZ    S_HL_5
        MOV   D, #0x01
S_HL_5: set1  [HL].5
        MOV   A, [HL]
        CMP   A, #00111111b
        BZ    S_HL_6
        MOV   D, #0x01
S_HL_6: set1  [HL].6
        MOV   A, [HL]
        CMP   A, #01111111b
        BZ    S_HL_7
        MOV   D, #0x01
S_HL_7: set1  [HL].7
        MOV   A, [HL]
        CMP   A, #11111111b
        BZ    S_HL_out
        MOV   D, #0x01

S_HL_out:
        ret

// subroutine to read by HL addressed ram, check bitwise if content is 1 and set bitweise to 0
r1_w0:
        MOV   A, [HL]           // move to be tested byte into reg A
        CMP   A, #11111111b
        BZ    C_HL_0
        MOV   D, #0x01
C_HL_0: clr1  [HL].0
        MOV   A, [HL]
        CMP   A, #11111110b
        BZ    C_HL_1
```

```
        MOV   D, #0x01
C_HL_1: clr1  [HL].1
        MOV   A, [HL]
        CMP   A, #11111100b
        BZ    C_HL_2
        MOV   D, #0x01
C_HL_2: clr1  [HL].2
        MOV   A, [HL]
        CMP   A, #11111000b
        BZ    C_HL_3
        MOV   D, #0x01
C_HL_3: clr1  [HL].3
        MOV   A, [HL]
        CMP   A, #11110000b
        BZ    C_HL_4
        MOV   D, #0x01
C_HL_4: clr1  [HL].4
        MOV   A, [HL]
        CMP   A, #11100000b
        BZ    C_HL_5
        MOV   D, #0x01
C_HL_5: clr1  [HL].5
        MOV   A, [HL]
        CMP   A, #11000000b
        BZ    C_HL_6
        MOV   D, #0x01
C_HL_6: clr1  [HL].6
        MOV   A, [HL]
        CMP   A, #10000000b
        BZ    C_HL_7
        MOV   D, #0x01
C_HL_7: clr1  [HL].7
        MOV   A, [HL]
        CMP   A, #00000000b
        BZ    C_HL_out
        MOV   D, #0x01
C_HL_out:
        ret


        END
```

```
/****************************************************/
/*                                                  */
/*                                                  */
/*  PROJECT      = 78K0 Selftest                    */
/*  DEVICE       = 78K0/KF2 (uPDF0547)              */
/*  TEST         = March X RAM Test                 */
/*                                                  */
/*               HL DE reg is used localy  -> Stack */
/*                                                  */
/*                                                  */
/****************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    stl_ram_marchxmb            ; Label this program
        PUBLIC  stl_ram_marchxmb

        RSEG  SADDR_Z(1)
marchx_lgth    DS   2
marchx_start   DS   2
marchx_stop    DS   2

        RSEG  CODE
//*************************************************************************
//      stl_ram_marchxmb
//
//      AX contains the start address of to be tested RAM area
//      BC contains the length of the to be tested RAM area
//
//      A  returns 0xFF if a memory falt was detected, otherwise 0x00 is returned.
//*************************************************************************

stl_ram_marchxmb:
        PUSH  DE
        PUSH  HL                // store HL register
        MOVW  S:marchx_start, AX// store start address of marchx test
        MOVW  HL, AX            // start address into HL
        MOVW  AX, BC            // test length into AX
        MOVW  S:marchx_lgth, AX // store test length
// test if length is zero
        SUBW  AX, #0x0001       // decrement test_length_value
        BNC   length_not_zero   // if test_length_value was zero, c flag is set
        MOV   A, #0x00          // return value true - no error
        POP   HL
        POP   DE
        ret
// start march -c test
length_not_zero:
        MOVW  AX, S:marchx_lgth // reload test length into AX
        XCH   A, X              // calculate march-c stop address
        ADD   A, L              // add lower byte
        XCH   A, X
        ADDC  A, H              // add higher byte
        DECW  AX                // adjust stop value ( n-1 )
        MOVW  S:marchx_stop, AX // store march-c end address
// step 1 - <>(w0) - clear specified memory area
        MOV   A, #0x00          // clear register A
        MOV   D, A// clear error indicator
loop_step_1:
        MOV   [HL], A           // clear [HL] content
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_2    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_1       // next cycle

// step 2 - >(r0,w1) - read bitweise if zero is in RAM, then write 1 into it, from lower address
march_c_step_2:
        MOVW  AX, S:marchx_lgth // march-c length into AX
```

```
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchx_start// start address into AX
        MOVW  HL, AX            // start address into HL
loop_step_2:
        CALL  r0_w1             // read 0, write 1 to by HL addressed byte
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_3    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_2       // next cycle

// step 3 - <(r1,w0) - read bitweise if one is in RAM, then write zero into it from upper address
march_c_step_3:
        MOVW  AX, S:marchx_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchx_stop // stop address into AX
        MOVW  HL, AX            // stop address into HL
loop_step_3:
        CALL  r1_w0             // read 1, write 0 to by HL addressed byte
        DECW  HL                // decrement RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    march_c_step_4    // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_3       // next cycle

// step 4 - <>(r0) - check if specified memory area is all zero
march_c_step_4:
        MOVW  AX, S:marchx_lgth // march-c length into AX
        MOVW  BC, AX            // march-c length into BC
        MOVW  AX, S:marchx_start// start address into AX
        MOVW  HL, AX            // start address into HL
loop_step_4:
        MOV   A, [HL]           // load RAM cell addressed by HL
        CMP   A, #0x00          // check if RAM content is zero
        BZ    is_zero
        MOV   D, #0x01          // D functions as error indicator
is_zero:
        INCW  HL                // increment RAM address register
        XCHW  AX, BC            // move BC ( length counter )  into AX
        SUBW  AX, #0x0001       // decrement length counter
        BZ    out_4             // check if all RAM is cleared ( check length counter )
        XCHW  AX, BC            // restore A ( 0x00 ) and BC ( length counter )
        BR    loop_step_4       // next cycle

// move test result into reg A and return it to calling function
out_4:
        MOV   A, D              // return test result
        POP   HL                // restore HL content
        POP   DE
        ret

// subroutine to read by HL addressed ram, check bitwise if content is 0 and set bitweise to 1
r0_w1:
        MOV   A, [HL]           // move to be tested byte into reg A
        CMP   A, #00000000b
        BZ    S_HL_0
        MOV   D, #0x01
S_HL_0: set1  [HL].0
        MOV   A, [HL]
        CMP   A, #00000001b
        BZ    S_HL_1
        MOV   D, #0x01
S_HL_1: set1  [HL].1
        MOV   A, [HL]
        CMP   A, #00000011b
        BZ    S_HL_2
        MOV   D, #0x01
S_HL_2: set1  [HL].2
        MOV   A, [HL]
        CMP   A, #00000111b
```

```
        BZ    S_HL_3
        MOV   D, #0x01
S_HL_3: set1  [HL].3
        MOV   A, [HL]
        CMP   A, #00001111b
        BZ    S_HL_4
        MOV   D, #0x01
S_HL_4: set1  [HL].4
        MOV   A, [HL]
        CMP   A, #00011111b
        BZ    S_HL_5
        MOV   D, #0x01
S_HL_5: set1  [HL].5
        MOV   A, [HL]
        CMP   A, #00111111b
        BZ    S_HL_6
        MOV   D, #0x01
S_HL_6: set1  [HL].6
        MOV   A, [HL]
        CMP   A, #01111111b
        BZ    S_HL_7
        MOV   D, #0x01
S_HL_7: set1  [HL].7
        MOV   A, [HL]
        CMP   A, #11111111b
        BZ    S_HL_out
        MOV   D, #0x01
S_HL_out:
        ret

// subroutine to read by HL addressed ram, check bitwise if content is 1 and set bitweise to 0
r1_w0:
        MOV   A, [HL]           // move to be tested byte into reg A
        CMP   A, #11111111b
        BZ    C_HL_0
        MOV   D, #0x01
C_HL_0: clr1  [HL].0
        MOV   A, [HL]
        CMP   A, #11111110b
        BZ    C_HL_1
        MOV   D, #0x01
C_HL_1: clr1  [HL].1
        MOV   A, [HL]
        CMP   A, #11111100b
        BZ    C_HL_2
        MOV   D, #0x01
C_HL_2: clr1  [HL].2
        MOV   A, [HL]
        CMP   A, #11111000b
        BZ    C_HL_3
        MOV   D, #0x01
C_HL_3: clr1  [HL].3
        MOV   A, [HL]
        CMP   A, #11110000b
        BZ    C_HL_4
        MOV   D, #0x01
C_HL_4: clr1  [HL].4
        MOV   A, [HL]
        CMP   A, #11100000b
        BZ    C_HL_5
        MOV   D, #0x01
C_HL_5: clr1  [HL].5
        MOV   A, [HL]
        CMP   A, #11000000b
        BZ    C_HL_6
        MOV   D, #0x01
C_HL_6: clr1  [HL].6
        MOV   A, [HL]
        CMP   A, #10000000b
        BZ    C_HL_7
        MOV   D, #0x01
```

```
C_HL_7: clr1  [HL].7
        MOV   A, [HL]
        CMP   A, #00000000b
        BZ    C_HL_out
        MOV   D, #0x01
C_HL_out:
        ret


        END
```

```
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = CPU Register Test                 */
/*                                                   */
/*                                                   */
/*****************************************************/
// Standard ICC78K include files


#include <io78f0547_80.h>

        NAME    cpu_register_test     ; Label this program
        PUBLIC  stl_registertest      ; funktion name

        RSEG  SADDR_Z(1)

errorindicator   DS8   1
#define errorflg errorindicator.0

        RSEG  CODE

//*************************************************************************
//      stl_registertest
//
//      Register content of Banks0, 1, 2, 3, have to be restored by this testroutine
//      it is assumed, the register test routine is entered with selected RB0
//
//      !!! in dependency of the user application it might be necessary to
//      dissable/enable interrupts while testing, because Register banks are
//      switched and original Register content is buffered on Stack
//
//*************************************************************************
stl_registertest:
//        DI                              // disable Interrupts necessary ???
        clr1    S:errorflg
        PUSH    AX                      // move on stack AX register content of RB0
        PUSH    BC                      // move on stack BC register content of RB0
        PUSH    DE                      // move on stack DE register content of RB0
        PUSH    HL                      // move on stack HL register content of RB0
        BR      go_tst                  // go to test routine

loop_rb1:
        SEL     RB1                     // Select register bank 1
        PUSH    AX                      // move on stack AX register content of RB1
        PUSH    BC                      // move on stack BC register content of RB1
        PUSH    DE                      // move on stack DE register content of RB1
        PUSH    HL                      // move on stack HL register content of RB1
        BR      go_tst
loop_rb2:
        POP     HL                      // restore HL register content of RB1
        POP     DE                      // restore DE register content of RB1
        POP     BC                      // restore BC register content of RB1
        POP     AX                      // restore AX register content of RB1
        SEL     RB2                     // Select register bank 2
        PUSH    AX                      // move on stack AX register content of RB2
        PUSH    BC                      // move on stack BC register content of RB2
        PUSH    DE                      // move on stack DE register content of RB2
        PUSH    HL                      // move on stack HL register content of RB2
        BR      go_tst
loop_rb3:
        POP     HL                      // restore HL register content of RB2
        POP     DE                      // restore DE register content of RB2
        POP     BC                      // restore BC register content of RB2
        POP     AX                      // restore AX register content of RB2
        SEL     RB3                     // Select register bank 3
        PUSH    AX                      // move on stack AX register content of RB3
        PUSH    BC                      // move on stack BC register content of RB3
        PUSH    DE                      // move on stack DE register content of RB3
        PUSH    HL                      // move on stack HL register content of RB3
```

```
// start register test routine
go_tst:
        MOVW    AX, #0xAAAA           // load AX with 0xAAAA
        CMPW    AX, #0xAAAA           // compare AX with 0xAAAA
        BZ      nxt_1                 // if not equal continue @ Error
        set1    S:errorflg
nxt_1:  MOVW    AX, #0x5555           // load AX with 0x5555
        CMPW    AX, #0x5555           // compare AX with 0x5555
        BZ      nxt_2                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_2:  MOVW    BC, #0xAAAA           // load BC with 0xAAAA
        MOVW    AX, BC                // move content of BC into AX, compare operation can only be done with AX
        CMPW    AX, #0xAAAA           // compare AX ( BC ) with 0xAAAA
        BZ      nxt_3                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_3:  MOVW    BC, #0x5555           // load BC with 0x5555
        MOVW    AX, BC                // move content of BC into AX, compare operation can only be done with AX
        CMPW    AX, #0x5555           // compare AX ( BC ) with 0x5555
        BZ      nxt_4                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_4:  MOVW    DE, #0xAAAA           // load DE with 0xAAAA
        MOVW    AX, DE                // move content of DE into AX, compare operation can only be done with AX
        CMPW    AX, #0xAAAA           // compare AX ( DE ) with 0xAAAA
        BZ      nxt_5                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_5:  MOVW    DE, #0x5555           // load DE with 0x5555
        MOVW    AX, DE                // move content of DE into AX, compare operation can only be done with AX
        CMPW    AX, #0x5555           // compare AX ( DE ) with 0x5555
        BZ      nxt_6                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_6:  MOVW    HL, #0xAAAA           // load HL with 0xAAAA
        MOVW    AX, HL                // move content of HL into AX, compare operation can only be done with AX
        CMPW    AX, #0xAAAA           // compare AX ( HL ) with 0x5555
        BZ      nxt_7                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
nxt_7:  MOVW    HL, #0x5555           // load HL with 0x5555
        MOVW    AX, HL                // move content of HL into AX, compare operation can only be done with AX
        CMPW    AX, #0x5555           // compare AX ( HL ) with 0x5555
        BZ      nxt_8                 // if not equal continue @ Error
        set1    S:errorflg            // if not equal continue @ Error
// check current Register Bank and select next one
nxt_8:  MOV     A, PSW                // test and select next register bank
        AND     A, #00101000B         // isolate RBS0 and RBS1
        CMP     A, #00000000B         // check if current RB is RB0
        BZ      loop_rb1              // if yes ( zero ) select RB1 and continue test
        CMP     A, #00001000B         // check if current RB is RB1
        BZ      loop_rb2              // if yes ( zero ) select RB2 and continue test
        CMP     A, #00100000B         // check if current RB is RB2
        BZ      loop_rb3              // if yes ( zero ) select RB3 and continue test
                                      // if no ( not zero ) selected RB is RB3 and test finished
        POP     HL                    // restore HL register content of RB3
        POP     DE                    // restore DE register content of RB3
        POP     BC                    // restore BC register content of RB3
        POP     AX                    // restore AX register content of RB3

        SEL     RB0                   // select register bank 0
        POP     HL                    // restore HL register content of RB0
        POP     DE                    // restore DE register content of RB0
        POP     BC                    // restore BC register content of RB0
        POP     AX                    // restore AX register content of RB0

//        EI                           // enable Interrupts, necessary ???
        BT      S:errorflg, out_error
        MOV     A, #0x00              // give back 00 ( no error ) to calling function
        ret
```

```
out_error:
        MOV     A, #0x01              // give back 01 ( error )to calling function
        ret

        ENDMOD

        END
```

```
/****************************************************/
/*                                                  */
/*                                                  */
/*  PROJECT      = 78K0 Selftest                    */
/*  DEVICE       = 78K0/KF2 (uPDF0547)              */
/*  TEST         = Cyclic Redundancy Check - CRC    */
/*                                                  */
/*               HL reg is used localy  -> Stack    */
/*                                                  */
/*                                                  */
/****************************************************/

// Standard ICC78K include files

#include <io78f0547_80.h>

        NAME    stl_crc             ; Label this program
        PUBLIC  stl_crc

        RSEG  SADDR_Z(1)
crc_lgth    DS   2
crc_input   DS   2

        RSEG  CONST

//*************************************************************************

// CRC16 CCITT Table

crc16_table:
        DW 0x0000,0x1021,0x2042,0x3063,0x4084,0x50a5,0x60c6,0x70e7,0x8108
        DW 0x9129,0xa14a,0xb16b,0xc18c,0xd1ad,0xe1ce,0xf1ef,0x1231,0x0210
        DW 0x3273,0x2252,0x52b5,0x4294,0x72f7,0x62d6,0x9339,0x8318,0xb37b
        DW 0xa35a,0xd3bd,0xc39c,0xf3ff,0xe3de,0x2462,0x3443,0x0420,0x1401
        DW 0x64e6,0x74c7,0x44a4,0x5485,0xa56a,0xb54b,0x8528,0x9509,0xe5ee
        DW 0xf5cf,0xc5ac,0xd58d,0x3653,0x2672,0x1611,0x0630,0x76d7,0x66f6
        DW 0x5695,0x46b4,0xb75b,0xa77a,0x9719,0x8738,0xf7df,0xe7fe,0xd79d
        DW 0xc7bc,0x48c4,0x58e5,0x6886,0x78a7,0x0840,0x1861,0x2802,0x3823
        DW 0xc9cc,0xd9ed,0xe98e,0xf9af,0x8948,0x9969,0xa90a,0xb92b,0x5af5
        DW 0x4ad4,0x7ab7,0x6a96,0x1a71,0x0a50,0x3a33,0x2a12,0xdbfd,0xcbdc
        DW 0xfbbf,0xeb9e,0x9b79,0x8b58,0xbb3b,0xab1a,0x6ca6,0x7c87,0x4ce4
        DW 0x5cc5,0x2c22,0x3c03,0x0c60,0x1c41,0xedae,0xfd8f,0xcdec,0xddcd
        DW 0xad2a,0xbd0b,0x8d68,0x9d49,0x7e97,0x6eb6,0x5ed5,0x4ef4,0x3e13
        DW 0x2e32,0x1e51,0x0e70,0xff9f,0xefbe,0xdfdd,0xcffc,0xbf1b,0xaf3a
        DW 0x9f59,0x8f78,0x9188,0x81a9,0xb1ca,0xa1eb,0xd10c,0xc12d,0xf14e
        DW 0xe16f,0x1080,0x00a1,0x30c2,0x20e3,0x5004,0x4025,0x7046,0x6067
        DW 0x83b9,0x9398,0xa3fb,0xb3da,0xc33d,0xd31c,0xe37f,0xf35e,0x02b1
        DW 0x1290,0x22f3,0x32d2,0x4235,0x5214,0x6277,0x7256,0xb5ea,0xa5cb
        DW 0x95a8,0x8589,0xf56e,0xe54f,0xd52c,0xc50d,0x34e2,0x24c3,0x14a0
        DW 0x0481,0x7466,0x6447,0x5424,0x4405,0xa7db,0xb7fa,0x8799,0x97b8
        DW 0xe75f,0xf77e,0xc71d,0xd73c,0x26d3,0x36f2,0x0691,0x16b0,0x6657
        DW 0x7676,0x4615,0x5634,0xd94c,0xc96d,0xf90e,0xe92f,0x99c8,0x89e9
        DW 0xb98a,0xa9ab,0x5844,0x4865,0x7806,0x6827,0x18c0,0x08e1,0x3882
        DW 0x28a3,0xcb7d,0xdb5c,0xeb3f,0xfb1e,0x8bf9,0x9bd8,0xabbb,0xbb9a
        DW 0x4a75,0x5a54,0x6a37,0x7a16,0x0af1,0x1ad0,0x2ab3,0x3a92,0xfd2e
        DW 0xed0f,0xdd6c,0xcd4d,0xbdaa,0xad8b,0x9de8,0x8dc9,0x7c26,0x6c07
        DW 0x5c64,0x4c45,0x3ca2,0x2c83,0x1ce0,0x0cc1,0xef1f,0xff3e,0xcf5d
        DW 0xdf7c,0xaf9b,0xbfba,0x8fd9,0x9ff8,0x6e17,0x7e36,0x4e55,0x5e74
        DW 0x2e93,0x3eb2,0x0ed1,0x1ef0



        RSEG  CODE
//*************************************************************************
//      stl_crc calculation
//
//      AX contains the start_crc_value
//      BC contains length of address range were crc should calculated from
//      DE contains the Address Pointer on first data were crc calculation should start from
//
//      AX contains the return value of new calculated crc_value
//*************************************************************************
```

```
stl_crc:
        PUSH  HL                // store HL register
// sort input parameter
        MOVW  S:crc_input, AX   // buffer crc input value for current calculation step
        MOVW  AX, BC            // move crc length into AX
        MOVW  S:crc_lgth, AX    // buffer crc length into crc_lgth
        MOVW  AX, S:crc_input   // after finished parameter sorting load crc input value again int AX
// start first calculation step
continue_crc_calc:
        MOVW  AX, S:crc_input   // after finished parameter sorting load crc input value again int AX
        XCH   A, X              // crc>>8 is now in register X
        MOV   A, [DE]           // load data of ROM/Flash area for new crc calculation into A
        XOR   X, A              // result of XOR is address value to catch correct table value
        XCH   A, X              // maske in A
        MOV   X, #0x00          // clear register X
        ROLC  A, 1              // multiplication * 2 of address offset value
        XCH   A, X              // table address offset value into X of AX register
        ROLC  A, 1              // multiplication * 2 of higher part of Address offset value
        ADDW  AX, #crc16_table  // add onto Table startaddress calculated offset
        MOVW  HL, AX            // calculated Table address into HL register
// load AX register with calculated table value
        MOV   A, [HL]           // move first byte ( low byte ) of crc16_table into A
        MOV   X, A              // move it into register X ( low byte )
        MOV   A, [HL+1]         // move second byte ( high byte ) of crc16_table into A
                                // AX contains now crc16 table value
        XCHW  AX, BC            // move crc16 table value into BC
        MOVW  AX, S:crc_input   // reload input crc value for this calculation step into register AX
        MOV   A, #0x00          // clear A, to prepare 8 times left shift of AX which means register X cleared
        XCH   A, X              // is equal 8 times left shift of AX registers
// XOR calculated crc16_table value with 8 times left shifted input crc value
        XOR   B, A              // XOR high bytes
        MOVW  AX, BC            // new CRC value into AX
        MOVW  S:crc_input, AX   // update crc input value to prepare next calculation step if not finished
// check if all data defined by crc_length have been calculated
        INCW  DE                // next ROM/Flash address for next crc-calculation
        MOVW  AX, crc_lgth      // move crc_lgth into AX
        SUBW  AX, #0x0001       // decrement AX ( crc_lgth )
        BZ    out               // if equal calculations finished
        MOVW  crc_lgth, AX      // store back crc_lgth
        BR    continue_crc_calc // continue

out:    MOVW  AX, S:crc_input   // calculations finished , move output parameter ( crc value ) into AX
        POP   HL                // restore HL register, which was modified inside this routine

        ret

        END
#include <stdio.h>
const unsigned int crc16_table[256] =
  {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7, 0x8108,
    0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef, 0x1231, 0x0210,
    0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6, 0x9339, 0x8318, 0xb37b,
    0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de, 0x2462, 0x3443, 0x0420, 0x1401,
    0x64e6, 0x74c7, 0x44a4, 0x5485, 0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee,
    0xf5cf, 0xc5ac, 0xd58d, 0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6,
    0x5695, 0x46b4, 0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d,
    0xc7bc, 0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b, 0x5af5,
    0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcbdc,
    0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87, 0x4ce4,
    0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41, 0xedae, 0xfd8f, 0xcdec, 0xddcd,
    0xad2a, 0xbd0b, 0x8d68, 0x9d49, 0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13,
    0x2e32, 0x1e51, 0x0e70, 0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a,
    0x9f59, 0x8f78, 0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e,
    0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e, 0x02b1,
    0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256, 0xb5ea, 0xa5cb,
    0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d, 0x34e2, 0x24c3, 0x14a0,
```

```
    0x0481, 0x7466, 0x6447, 0x5424, 0x4405, 0xa7db, 0xb7fa, 0x8799, 0x97b8,
    0xe75f, 0xf77e, 0xc71d, 0xd73c, 0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657,
    0x7676, 0x4615, 0x5634, 0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9,
    0xb98a, 0xa9ab, 0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882,
    0x28a3, 0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92, 0xfd2e,
    0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9, 0x7c26, 0x6c07,
    0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1, 0xef1f, 0xff3e, 0xcf5d,
    0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8, 0x6e17, 0x7e36, 0x4e55, 0x5e74,
    0x2e93, 0x3eb2, 0x0ed1, 0x1ef0,
  };


unsigned int fast_crc16 (unsigned int sum,  unsigned int len, unsigned char *p)
{
    while (len--){
      sum = crc16_table[(sum >> 8) ^ *p++] ^ (sum << 8);

    }
    return sum;
}
/*****************************************************/
/*                                                   */
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = Main Program - Header File        */
/*                                                   */
/*****************************************************/

// declaration functions used in Main program
void PeripheralInit(void);
void InitOscillator(void);
extern void DelayTimer_51(void);
void indicate_test_result(bool);
```

```
/*****************************************************/
/*                                                   */
/*  PROJECT      = 78K0 Selftest                     */
/*  DEVICE       = 78K0/KF2 (uPDF0547)               */
/*  TEST         = Header File                       */
/*                                                   */
/*****************************************************/

// declaration functions etc used in "stl" functions

//  #ifndef STL_H_INCLUDED
//  #define STL_H_INCLUDED

/*****************************************************************************************/
/*                                                                                       */
/*  User Definition Part                                                                 */
/*                                                                                       */
/*****************************************************************************************/

#define start_ram 0xFB20    // start ram address
#define end_ram 0xFEDF      // end ram address
#define start_rom 0x0000    // start rom address
#define end_rom 0xBFFF      // end rom address
#define rom_cs  0xAA        // reference check sum for rom test

/*****************************************************************************************/
/*                                                                                       */
/*  Self Test Library Definition Part - do not modify                                    */
/*                                                                                       */
/*****************************************************************************************/

// The following offsets must match with the INTCHK definition
#define INTCHK_FREQ             0       // offset to pointer to array of frequency counters
#define INTCHK_TICKS            2       // offset to pointer to array of ticks counters
#define INTCHK_FREQ_LOWER       4       // offset to pointer to array of lower frequencies
#define INTCHK_FREQ_UPPER       6       // offset to pointer to array of upper frequencies
#define INTCHK_FREQ_INITIAL     8       // offset to pointer to array of initial frequencies
#define INTCHK_TICKS_INITIAL   10       // offset to pointer to array of initial ticks
#define INTCHK_ARRAY_SIZE      12       // offset to number of frequencies, i.e. size of each previous array

#ifdef __IAR_SYSTEMS_ICC__

// TYPEDEF
typedef struct {unsigned char data[2] ; } COMPBYTE ;
typedef struct {unsigned int data[2] ; } COMPWORD ;
typedef struct {unsigned char data[2] ; } ECCBYTE ;
typedef struct {unsigned int data[2] ; } ECCWORD ;

typedef struct {      unsigned int *freq ;              // pointer to array of frequency counters
                      unsigned int *ticks ;             // pointer to array of ticks counters
                const unsigned int *freq_lower ;        // pointer to array of lower frequencies
                const unsigned int *freq_upper ;        // pointer to array of upper frequencies
                const unsigned int *freq_inital ;       // pointer to array of initial frequencies
                const unsigned int *ticks_inital ;      // pointer to array of initial ticks
                      unsigned int array_size ;         // number of frequencies, i.e. size of each previous array
               } INTCHK ;

// EXTERNAL VARIABLES
extern unsigned int ecc_detectederrors ; /* counter for detected errors in ECC functions */
extern unsigned int ecc_fixederrors ; /* counter for fixed errors in ECC functions */
extern unsigned int comp_detectederrors ; /* counter for detected errors in complement functions */

// FUNCTION PROTOTYPES
//unsigned int (*comp_errorhandler) (void *addr, unsigned char size, unsigned int data) ; /* pointer to error handler for
complement functions */
//unsigned int (*ecc_errorhandler) (void *addr, unsigned char size, unsigned char data) ; /* pointer to error handler for
ECC functions */
//unsigned char stl_control ; /* controls operation of the self test library; */

// EXTERNAL FUNCTION PROTOTYPES
```

```
extern void stl_init(void);
extern bool stl_registertest (void);
extern bool stl_pctest (void);
extern bool stl_clocktest (void);
extern unsigned char stl_checksum(unsigned char *addr, unsigned int size) ;
extern bool *stl_ram_cb_testb(unsigned char *addr, int num);
extern unsigned char stl_romtest (int , int , char);
extern unsigned int comp_error(void *addr, unsigned char size, unsigned int data) ;
extern unsigned int ecc_error(void *addr, unsigned char size, unsigned char data) ;
extern unsigned char comp_bread(COMPBYTE *addr) ;
extern unsigned int comp_wread(COMPWORD *addr) ;
extern unsigned char ecc_bread(ECCBYTE *addr) ;
extern unsigned int ecc_wread(ECCWORD *addr) ;
extern unsigned int stl_crc(unsigned int crc, unsigned int length, unsigned char *data);
extern unsigned int fast_crc16(unsigned int sum, unsigned int len, unsigned char *p);
extern bool stl_ram_marchcmb(unsigned char *addr, unsigned int num);
extern bool stl_ram_marchxmb(unsigned char *addr, unsigned int num);
extern bool stl_inttest(INTCHK *intchk) ;
extern COMPBYTE *comp_bwrite(COMPBYTE *addr, unsigned char data) ;
extern COMPWORD *comp_wwrite(COMPWORD *addr, unsigned int data) ;
extern ECCBYTE *ecc_bwrite(ECCBYTE *addr, unsigned char data) ;
extern ECCWORD *ecc_wwrite(ECCWORD *addr, unsigned int data) ;

#endif /* __IAR_SYSTEMS_ICC__ */

// #endif
```

```
//*****************************************************************************
//
//      Self Test Library
//
//*****************************************************************************
//      stl_global_data_example
//
//     used for main excample
//
//*****************************************************************************

//#include <intrinsics.h>
//#include <io78f0547_80.h>

// Option Byte settings
#pragma constseg = OPTBYTE
__root const char option [5]  = {0x6E,0x00,0x00,0x00,0x00};
#pragma constseg = default

// definitions used in main for Self Test Funktions
int i;
int I;
char result;
bool resultBool;

// RAM tests
// unsigned char RAM_TEST_results
unsigned char *p_MC_StAdr;
unsigned int RAM_MarchC_lgth;
unsigned char *p_MX_StAdr;
unsigned int RAM_MarchX_lgth;
unsigned char *addr;
int num;
int size;

// return values from ecc/comp test;
unsigned char eb ;
unsigned int ew ;
unsigned char cb ;
unsigned int cw ;

// CRC development interim
unsigned int crc;
unsigned char *pStAdr;
unsigned int crc_length;

// Error correction variables definitions
COMPBYTE cb1 ;
COMPWORD cw1 ;
ECCBYTE eb1 ;
ECCWORD ew1 ;
ECCBYTE *dummy;
```