CANopen Firmware Framework for the ELMB

(Embedded Local Monitor Board)

Version 1.1

(ELMB128, approximate true size)

Henk Boterenbrood
(*boterenbrood@nikhef.nl*)
NIKHEF, Amsterdam
10 Mar 2004

Contents

Version History		
Version	**Date**	**Comments**
1.1	10 Mar 2004	Major update.
1.0	7 Mar 2003	First version.
0.0	Mar 2003	Draft version.

Table 1. Document change record.

1 Introduction

1.1 Hardware

The ***Embedded Local Monitor Board*** (**ELMB**) is a plug-on board designed for the ATLAS experiment, where it will be used for a range of different control and monitoring tasks.
It was designed with low-power, low-cost and high I/O-channel density in mind. Tolerance to radiation is another very important issue, and extensive radiation tests have been carried out to determine the ELMB's (non-)sensitivity to TID, NIEL and SEE.

The ELMB contains an 8-bit microcontroller, a CAN-controller and CAN-bus driver. Optionally the ELMB is equipped with a 16-bit ADC (Crystal CS5523) and multiplexor circuitry for 64 analog inputs.
The microcontroller is *In-System-Programmable* via an onboard connector, but is also *In-Application-Programmable* via the CAN-bus, enabling true *remote* firmware upgrades.
A block diagram of the ELMB is shown in Figure 1.

The ***CAN*** bus is the chosen fieldbus by the ATLAS *Detector Control System* (DCS) for interconnecting distributed I/O within the detector. The ***CANopen*** protocol [1] has been adopted as the communication protocol standard to be used on the CAN-bus.

The latest version of the ELMB is called **ELMB128**, based on the 8-bit ATMEL ***ATmega128*** **microcontroller** and runs at a clockspeed of 4 MHz. Older versions of the ELMB are based on the **ATmega103** microcontroller, which is the predecessor of the ATmega128. These two microcontroller types are highly compatible, but the ATmega128 offers extended features, the self-programming feature being the most important one (removing the need for a second microcontroller which was present on older types of ELMB).
The microcontroller has 128 Kbyte of Flash-memory, 4 kByte of SRAM, 4 Kbytes of EEPROM and a number of on-chip peripherals including 4 timer/counters and general-purpose I/O pins.

The onboard CAN-controller is the Infineon 81C91, a socalled 'Full-CAN Controller' with buffers for 16 different messages.

The board is also fitted with opto-couplers to decouple the board from CAN-bus and the analog part with ADC (if present).

Note that to control the ADC the ADC's *SPI* interface (3 lines) as well as 2 extra lines, a chip-select signal (***CS***) and a socalled 'latch' signal (***Latch***) are necessary, requiring 5 lines in total. These must be taken from the ELMB's general-purpose lines available and routed via a motherboard to the ADC, if the ELMB microcontroller must control the ADC.
Also note that the 3 digital I/O lines (that also constitute an SPI interface on the microcontroller) are used to control the CAN-controller, and should thus be used with care (external devices controlled through it should have a select/deselect-signal).

Figure 1. Block diagram of the ELMB128 module. Of the 37 Dig I/O lines 5 would be needed to control the ADC.

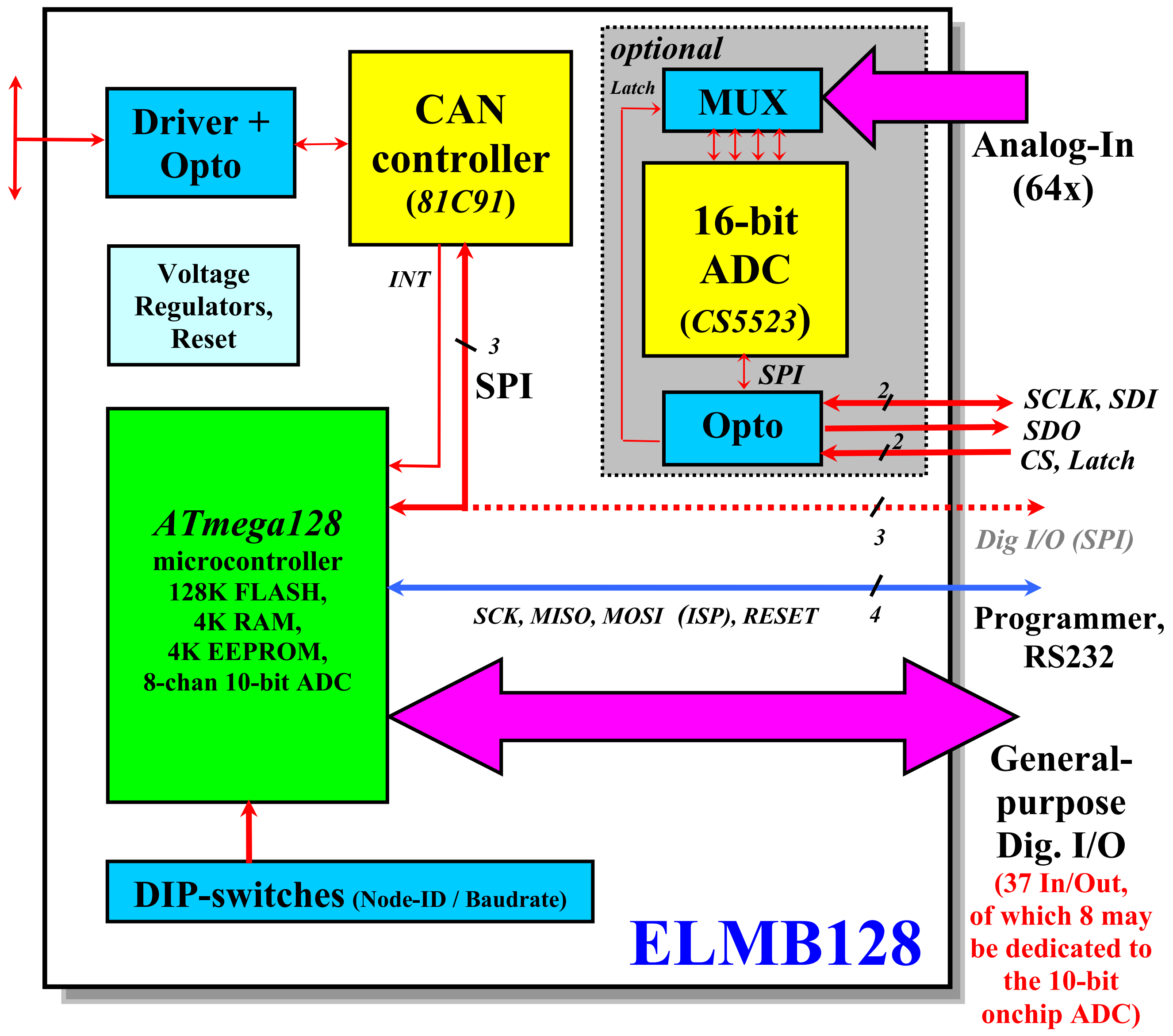
Driver + Opto
CAN controller (81C91)
optional
Latch
MUX
Analog-In (64x)
16-bit ADC (CS5523)
Voltage Regulators, Reset
INT
3
SPI
SPI
Opto
2
2
SCLK, SDI
SDO
CS, Latch
ATmega128 microcontroller 128K FLASH, 4K RAM, 4K EEPROM, 8-chan 10-bit ADC
3
Dig I/O (SPI)
SCK, MISO, MOSI (ISP), RESET
4
Programmer, RS232
General-purpose Dig. I/O
(37 In/Out, of which 8 may be dedicated to the 10-bit onchip ADC)
DIP-switches (Node-ID / Baudrate)
ELMB128

Table 2 shows a list of the microcontroller's I/O port pins and their availability to the ELMB user.

In addition to the I/O pins in the table the microcontroller pins *ALE*, *RD_* and *WR_* are externally available, enabling the use of external RAM (up to 64 kByte) using PORTA and PORTC as addressbus/databus. Alternatively these pins can be used as general-purpose I/O-pins (Port G, see ATmega128 datasheet).

I/O PORT: **Function:**	**A** **In/Out**	**B** **In/Out**	**C** **In/Out**	**D** **In/Out**	**E** **In/Out**	**F** **I/O/ADC**
pin 0	*a*	—	*a*	— (*i*)	—	*a*
pin 1	*a*	*SCLK*	*a*	— (*i*)	—	*a*
pin 2	*a*	*SDI*	*a*	— (*i*)	—	*a*
pin 3	*a*	*SDO*	*a*	*a*(*i*)	*a*	*a*
pin 4	*a*	—	*a*	*a*	*a*(*i*)	*a*
pin 5	*a*	—	*a*	*a*	*a*(*i*)	*a*
pin 6	*a*	—	*a*	*a*	*a*(*i*)	*a*
pin 7	*a*	—	*a*	*a*	*a*(*i*)	*a*

Table 2. ELMB microcontroller (ATmega128) I/O pins, their function and/or availability for user applications.

—	=	*NOT* freely available (see Table 3).
a	=	available for general use.
(*i*)	=	has external interrupt capability.
SCLK/SDI/SDO	=	lines carrying **SPI**-protocol for communication with the onboard CAN-controller.

In addition to the port pins shown in the table the ELMB128 has 3 extra I/O pins available on its external connectors: **G0**, **G1** and **G2** (marked on the ELMB external connector layout as **WR**, **RD** and **ALE**).

Table 3 below lists the functions of the microcontroller I/O pins not freely available for user purposes (or with restrictions). They are for ELMB-specific purposes, such as reading the DIP-switches, operating the CAN-controller and connecting to an external Programmer/RS232-interface. This is important information only for developers of ELMB software.

I/O PORT	**B**	**D**	**E**
pin 0	DIP-8/DIP-2	Slave RESET	Slave-ISP data out, ISP data out, RXD0
pin 1	*SCLK*, DIP-7/DIP-1, ISP clock in	Slave-to-Master, CAN interrupt, Slave-ISP clock out	Slave-ISP data in, ISP data in, TXD0
pin 2	*SDI*, DIP-6	Master-to-Slave, Slave-to-Master	enable read DIP-1 to 2
pin 3	*SDO*, DIP-5		
pin 4	DIP-4		
pin 5	CAN rd/wr, DIP-3		
pin 6	CAN chip select		
pin 7	enable read DIP-3 to 8		

Table 3. ELMB-specific microcontroller I/O pin functions.
DIP-*x* = DIP-switch *x*.
ISP = In-System-Programming, via the Programmer connector.
TXD0/RXD0 = serial output/input lines, via the Programmer connector.
Greyed-out functions are obsolete ('Slave' refers to the secondary microcontroller which is present on older types of ELMB).
To read DIP-3 to DIP-8 switches PORTB pin 7 must be set low. To read DIP-1 and DIP-2 switches PORTE pin 2 must be set low.
(the framework software described in the next sections uses DIP-1 and DIP-2 for the CAN-bus baudrate setting and DIP-3 to DIP-8 for the module's Node-ID)

Further and full details about the ELMB schematics, hardware and radiation tests can be obtained from the ELMB webpages[1].

[1] http://elmb.web.cern.ch/

1.2 Software

The rest of this document describes a *CANopen* firmware framework for the **ELMB** and **ELMB128**, developed for developers of ELMB custom application software.

The framework consists of a set of source code files that comprise a fully functional CANopen application for the ELMB's ATmega microcontroller. It is easily extendible by ELMB users who want to develop custom application firmware. They may use this framework as a starting point and can extend, modify and add to the existing source code and focus on the implementation of their front-end Detector Control System application, because many of the details of the CAN and CANopen operations are handled by the framework.

The framework *as-is* is a ready-to-run application, which can be compiled for both types of ELMB (i.e. with or without the secondary so-called 'Slave' processor which on the ELMB103 performs the firmware-upgrade and watchdog functions), by means of compile-time defines, as listed in the 'Compile Options' table in section 4.

The framework firmware provides:

- CANopen Network Management (NMT)
- a CANopen SDO server for reading and writing the Object Dictionary (expedited message transfer only)
- a basic Object Dictionary (see section 4)
- a CANopen PDO skeleton for 4 TPDOs and 4 RPDOs
- CANopen Nodeguarding/Heartbeat
- CANopen Lifeguarding
- Configuration storage in EEPROM
- access to Serial Number and ADC calibration constants (data stored in EEPROM during the ELMB production)
- interaction with and reprogramming of the Slave processor (only for the older ELMB version with ATmega103 microcontroller)
- Timer functions for busy-wait delays, time-outs and periodic actions
- Watchdog timer support
- ELMB CANopen Bootloader support
- Several mechanisms to improve the application's radiation-tolerance, some of which are optional at compile time (can be switched off when radiation-tolerance is not an issue).
- CAN-controller library based on interrupt-based CAN-message reception and message buffering (up to 64 messages; no priority scheme implemented for this buffer)

2 Using the Framework

2.1 Introduction

The framework files include a project file for the ICCAVR development toolset. Open the project's 'Options' dialog to set the paths to match your installation. Also check the 'Macro Defines' and 'Undefines' in the 'Compiler' tab of the 'Options' dialog, to see if they match your requirements (for a description of the 'defines' see Table 5 in section 4).

Each ELMB module shipped comes programmed with a general-purpose CANopen application called ***ELMBio***, which is described elsewhere [2]. Each ELMB also contains a Bootloader program, enabling code download via the CAN-bus, so if you have a CAN interface in your PC, it is easy to reprogram the ELMB with the framework application or your custom application, using an ELMB download tool.

For documentation on the ***ELMBio*** application and download tool(s) for the ELMB, go to the ELMB software webpage [3].

The first thing you do to develop software for the ELMB is to write the low-level functions for your specific hardware and to test these with some test program that doesn't use CAN, but for example simply interacts via the ELMB's serial port with the user.
Once you are confident that your low-level functions are correct you can integrate them into the CANopen framework and your application built on top of it. The low-level functions you developed are typically in a file you then should be able to use unchanged in your CANopen application.

The way to start developing the actual ELMB CANopen application is to make a list of data objects you want to monitor or read-out and parameters and settings you want to be able to read and write to. Then you map this list into objects that can be added to the CANopen Object Dictionary of your application (see section 4).
Note that the framework only supports so-called Expedited Transfers for SDOs, meaning that data objects must be 4 bytes or less in size.
At this point you could already write code to access your objects with SDO messages, because once you have this you can interactively control your hardware just using SDO messages, for which simple low-level PC tools are available.

2.2 Adding Your Code

The main entry point for adding code to the framework application software is in files `app.c` and `app.h`. Some comments are added to the various functions in `app.c` to guide you further. Most of the functions need filling in, but all of them are functional as-is within the framework and demonstrate their basic functionality when running the framework as-is.

Note at the top of the file these lines:

```
/* Include the functions that access your (custom) hardware */
//#include "your_lowlevel_hardware_functions.h"
```

It is assumed you will have another file `your_lowlevel_hardware_functions.c` containing the low-level functions that deal with the implementation details of your hardware. The functions in `app.c` will use these low-level functions to implement their own functionality that is more at the CANopen level.

The following functions have been defined in `app.c` :

- `app_init()`
  initialises your hardware, including read-out of saved settings.
- `app_status()`
  returns a number of bytes containing status information about your hardware, etc.; meant to be returned when reading Object 1002 (*Manufacturer Status Register*).
- `app_rpdo1()`
  processes the data bytes received in an RPDO1 message.
- `app_rpdo2()`
  processes the data bytes received in an RPDO2 message.
- `app_rpdo3()`
  processes the data bytes received in an RPDO3 message.
- `app_rpdo4()`
  processes the data bytes received in an RPDO4 message.
- `app_tpdo1()`
  starts up a multi-channel read-out sequence by making a call to `app_tpdo_scan_start()` resulting in multiple TPDO1 messages containing the proper data, obtained from your hardware (this function could also just generate a single message; here TPDO1 has been made a multi-channel PDO as an example only); subsequent repeated calls of `tpdo_scan()` by the main loop (in `ELMBmain.c`, when the node state is *Operational*) will call –among others– function `app_tpdo_scan()` which in its turn calls `app_scan_next()` which does the actual data gathering and PDO message creation and sending, one-by-one in subsequent calls.
- `app_tpdo2()`
  produces a TPDO2 message containing the proper data (obtained from your hardware).
- `app_tpdo3()`
  produces a TPDO3 message containing the proper data (obtained from your hardware).
- `app_tpdo4()`
  produces a TPDO4 message containing the proper data (obtained from your hardware).
- `app_tpdo_on_cos()`
  produces TPDO messages depending on the occurrence of a change-of-state of the application (to be defined by you); in the example code TPDO2 is produced but it could be any other TPDO or even multiple TPDOs; this function is called by tpdo_scan() which is called by the main loop (in ELMBmain.c, when the node state is *Operational*).
- `app_tpdo_scan_start()`
  sets a multi-channel TPDO read-out sequence in motion (see `app_tpdo1()`)
- `app_tpdo_scan_stop()`
  does everything necessary to stop an ongoing multi-channel TPDO read-out sequence; is called e.g. when the node is taken out of state *Operational*.

- `app_tpdo_scan()`
  is called in the main application loop to handle any ongoing multi-channel TPDO read-out sequences.
- `app_scan_next()`
  checks for the availability of data for the next channel in a multi-channel TPDO read-out sequence, fills a message buffer with the data and sends the CAN message.
- `app_sdo_read()`
  called by the SDO server (in `sdo.c`) to handle SDO read requests for objects in the Object Dictionary concerning your application data and parameters; the current code in this function is for demonstration purposes only; new object identifiers are preferably to be added to file `objects.h`.
- `app_sdo_write()`
  called by the SDO server (in `sdo.c`) to handle SDO write requests for objects in the Object Dictionary concerning your application data and parameters; the current code in this function is for demonstration purposes only; new objects identifiers are preferably to be added to file `objects.h`.
- `app_get_par()`
  called by `app_sdo_read()` to read data items from a particular object; the current code in this function is for demonstration purposes only; there could be several functions of this type in your code that each deal with different objects, all dependent on how you structure your Object Dictionary and how data items are to be obtained.
- `app_store_config()`
  stores up to 16 bytes of configuration parameters in permanent storage in EEPROM; define the number of bytes to store as APP_STORE_SIZE, in the source code defined just above this function; is called when the appropriate CANopen message to store parameters is received; the saved settings are read from EEPROM at every power-up or reset (see `app_load_config()` below`).`
  Note the difference between this type of storage and the storage of so-called 'working copies' of certain (global) variables which can be enabled or disabled in the code by defining *_VARS_IN_EEPROM_*, which serves to improve the rad-tolerance of the running program and not to save your settings between resets. Variable `AppChans` demonstrates its use.
- `app_load_config()`
  reads the saved settings from EEPROM and uses them to initialize your current settings; make sure to match the databytes saved by function `app_store_config()` and read by function `app_load_config().`

Depending on your requirements it might be necessary to make changes and additions to other files as well, for example if you want:

- to add new objects to the dictionary: `objects.h`
- to change the number of bytes in a particular PDO message: `can.h`
- to change the version string of your application: `1XXconf.h`
- to add 'clients' for Timer0 time-out services (see below): `timer1XX.h`
- to use a sofar unused interrupt source, replace the default interrupt handler for it: `intrpt.c`
- to store a new block of configuration parameters or to add run-time variables for which a copy is kept in EEPROM (for increased rad-tolerance): `store.c` and `store.h`

To download new code to your ELMB we assume it is equipped with a Bootloader, either in the ATmega128 processor (on the ELMB128), or in the shape of a second microcontroller (on older ELMB103 types). The new code can be downloaded via the CAN-bus when the Bootloader is running.

There are 2 alternative ways to make the Bootloader the currently running ELMB application:

- power the ELMB off and then on; the Bootloader is now in control: it sends a CANopen Bootup message (to be removed in a next Bootloader version), followed by a specific Emergency message (see the list of Emergency Objects in section 5); the Bootloader automatically jumps to the main application program (if it detects one) after 4 seconds, unless you send it within those 4 s any CAN-message it is programmed to handle; it then remains active and is ready to receive further programming instructions.
- if a CANopen application is in control of the ELMB (either the original **ELMBio** application, the framework program or your framework-based CANopen application), you can write to Object 5C00 (see Object Dictionary in section 4) to force the application to jump to the Bootloader (now it does *not* automatically jump back to the main application after 4 s).

2.3 Some Useful General-Purpose Functions

2.3.1 Timer Functions

The ELMB's ATmega128 microcontroller has 4 timers/counters (ATmega103 has 3 timers/counters). The framework uses Timers 0, 1 and 2, so Timer 3 is free for other purposes. The timer functions can be found in files `timer0.c`, `timer1.c` and `timer2.c`, and the constants and prototypes in `timer1XX.h`.

Timer0 is used for general-purpose time-outs on operations. It is configured to provide a clocktick of 10 ms, so it works well for time-outs > 10 ms. A time-out can be set on any number of independent operations, by assigning a 'client-identifier' to each operation, a number between 0 and the maximum number of 'clients'. This maximum number of time-out clients T0_CLIENTS must be set in `timer1XX.h` (is defined as 1 in the framework code).

Timer1 is used for not-so-frequent periodic operations. It is configured to provide a clocktick of 1 s. It is used for triggering Lifeguarding and Heartbeat operations, periodic PDO transmissions and Watchdog Timer resets.

Timer2 is used for busy-wait delays of 10 μs and up (to 63 ms). For busy-wait delays < 10 μs use calls to '*NOP( )*' (see file `general.h`): each *nop*-instruction ('*no operation*') takes 0.25 μs, with the ATmega128 clocked at 4 MHz.

The listing below describes some of the general-purpose timer functions made available through include file `timer1XX.h`.

---

void timer0_set_timeout_10ms(BYTE client, BYTE ticks)

```
Inputs
     client : client identifier
     ticks  : number of 10 ms ticks
Outputs
     none
Return value
     none
Description
     Set a time-out of ticks times 10 ms for a client with identifier
     client (add client identifiers in timer1XX.h).
     Time-out status to be checked by calling timer0_timeout(client).
     Parameter ticks can have any value up to 255.
     The actual time-out value t in ms is 10*(ticks-1)<=t<=10*ticks, so
     ticks should always be chosen 1 larger than the actual required
     minimum time-out, and only ticks >= 2 should be chosen; a maximum
     time-out of 255*10 ms = 2.55 s can be set.
```

BOOL timer0_timeout(BYTE client)

```
Inputs
     client : client identifier
Outputs
     none
```

Return value
Boolean which is true if a time-out has occurred.
Description
Must be called periodically to check if the time-out previously set by *timer0_set_timeout_10ms( )* has been reached.

---

void timer2_delay_mus(BYTE microseconds)

Inputs
microseconds: number of microseconds of required delay
Outputs
none
Return value
none
Description
Busy-wait delay of *microseconds* µs.
This routine is suitable for a delay of an even number of microseconds from 10 up to 256 microseconds (an odd number of microseconds gets rounded to the next even number).
NOTE: the overhead of this routine is about 7 microseconds (at 4 MHz), which is taken into account to achieve the desired delay.

---

void timer2_delay_ms(BYTE milliseconds)

Inputs
milliseconds: number of milliseconds of required delay
Outputs
none
Return value
none
Description
Busy-wait delay of *milliseconds* ms (actually 1.024 ms).
This routine is suitable for a delay of 1 up to 63 milliseconds only.

2.3.2 EEPROM Functions

File `eeprom.c` contains EEPROM byte-read and byte-write functions. Note that part of the EEPROM is reserved by the framework for various purposes. See the next section for details.

Next follows a description of the EEPROM functions made available through include file `eeprom.h`.

BYTE eeprom_read(BYTE addr)

```
  Inputs
      addr    : EEPROM address
  Outputs
      none
  Return value
      data byte read from EEPROM address addr.
  Description
      Read a byte from ATmega128/ATmega103 EEPROM from an address up to
      255 (FFh).
```

---

void eeprom_write(BYTE addr, BYTE byt)

```
  Inputs
      addr    : EEPROM address
      byt     : byte value to be written
  Outputs
      none
  Return value
      none
  Description
      Write byte value byt to ATmega128/ATmega103 EEPROM address addr;
      for addresses up to 255 (FFh).
```

BYTE eepromw_read(UINT16 addr)

```
  Inputs
      addr    : EEPROM address
  Outputs
      none
  Return value
      data byte read from EEPROM address addr.
  Description
      Read a byte from any address in ATmega128/ATmega103 EEPROM
      (up to address 4095 (FFFh)).
```

---

void eepromw_write(UINT16 addr, BYTE byt)

```
  Inputs
      addr    : EEPROM address
      byt     : byte value to be written
  Outputs
      none
  Return value
      none
  Description
      Write byte value byt to ATmega128/ATmega103 EEPROM address addr;
      for all addresses (up to 4095 (FFFh)).
```

2.4 EEPROM Memory Map

Table 4 below details the layout of the ELMB's EEPROM in the framework application. Read the comment in files `store.h` and `store.c` for more details.

EEPROM	ADDR	DESCRIPTION
not used	0000	
ELMBfw configuration parameters	0001 00A0	Holds permanently saved application configuration and settings, stored in up to 8 blocks of up to 16 bytes each; includes a CRC checksum for each data block.
Rad-tolerant working copy of global settings and parameters	00A1 00E0	Holds a copy of most application configuration and settings and some other parameters that don't change very often; parameters are reread from EEPROM each time before being used; this is an optional feature to counter the effects of SEE (Single Event Effects)
not used	00E1 00FF	
ELMB Serial Number	0100 0106	Holds the ELMB Serial Number given to it at production time; serves to uniqely identify the ELMB and retrieve its calibration constants and/or production data in the offline database.
reserved	0107	
not used	0108 011F	
ELMB Analog-in calib consts	0120 01CF	Holds the calibration constants, which were determined at production time, for all 6 voltage ranges (note: only present for ELMBs with an analog input part).
not used	01E0 0FFF	Free for other purposes (3616 bytes).

Table 4. EEPROM memory map within the *ELMBfw* framework application.

2.5 List of Files

…list of files and per file short description of content…to be done…

3 How it works

3.1 Setting CAN Node Identifier and Baudrate

Using the ELMB's onboard DIP-switches a node identifier can be set between 1 and 63 (has to be unique on the CAN-bus the board is on), using 6 of the 8 switches, and a CAN-bus baudrate of 50, 125, 250 or 500 Kbit/s, using the 2 remaining switches. See Figure 2 below for details. (Note: if your ELMB contains Bootloader version 1.3 or later, it is possible to implement a remotely configurable, i.e. via CAN messages, node identifier, an identifier which is no longer taken from the DIP-switch settings but stored in and read from the ELMB's EEPROM; for details contact author).

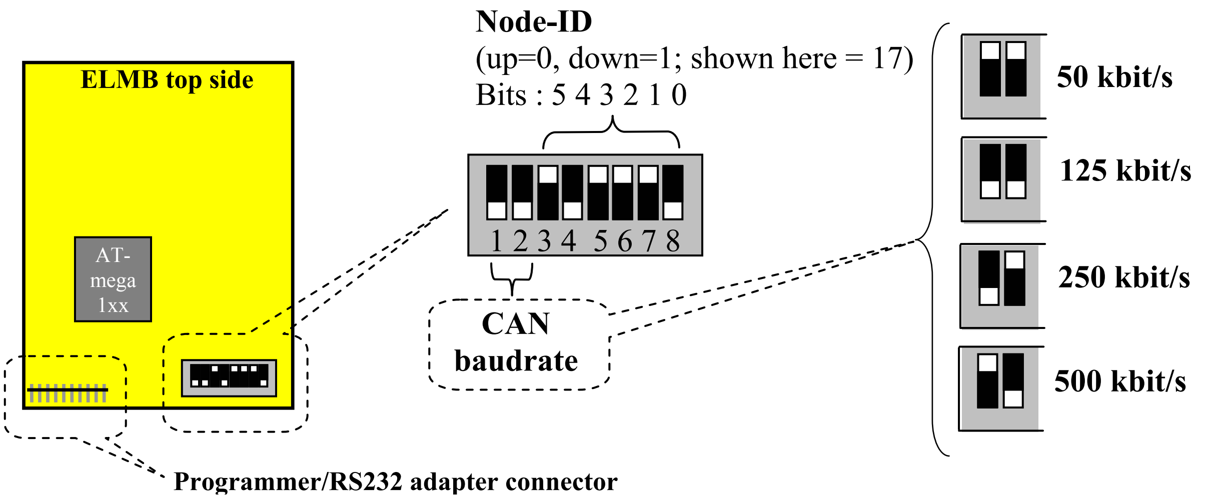

Figure 2. Location and function of ELMB DIP-switches and programming connector.

Figure 1 also indicates the connector to which the special *Programmer-and-RS232-adapter* (schematic available [1]) is connected, enabling serial *In-System-Programming* of the ELMB by e.g. a host PC and also enabling RS232 output by the ELMB ATmega128 processor (for development and test purposes).

3.2 Initialisation

After power-up, watchdog reset, manual reset or a *CANopen* initiated reset action (i.e. by an NMT *Reset-Node* message, see below) a *CANopen* node sends a socalled ***Boot-up*** message (as defined by the *CANopen* standard) as soon as it has finished initialising (hardware, software); this is a CAN-message with the following syntax:

ELMB (NMT-Slave) $\longrightarrow$ Host (NMT-Master)

COB-ID	DataByte 0
700h + *NodeID*	0

[1] see **http://atlasinfo.cern.ch/ATLAS/GROUPS/DAQTRIG/DCS/LMB/SB/index.html**

NodeID is the CAN node identifier set by means of the ELMB onboard DIP-switches, which according to the *CANopen* standard must be in the range between 1 and 127 and in the framework can be to set to a value between 1 and 63, as shown in Figure 2.

To *start* the application in the *CANopen* sense of the word, the following *CANopen* NMT (*Network ManagemenT*) message must be sent:

Host (NMT-Master) → ELMB (NMT-Slave)

COB-ID	DataByte 0	DataByte 1
000h	1 (*Start_Remote_Node*)	*NodeID* or 0 (all nodes on the bus)

There is no reply to this message.

Now the application is *Operational*, meaning that it monitors I/O channels as required and sends and receives (and processes) ***PDO*** messages (carrying the application data)), depending on the specific application.

Optionally a feature called *auto-start* may be enabled, so that the application automatically goes to *Operational* state after power-up or reset. The *auto-start* feature can be configured in *OD* index 3200h, subindex 2.

To generate a soft reset to the application the following *CANopen* NMT message must be sent:

Host (NMT-Master) → ELMB (NMT-Slave)

COB-ID	DataByte 0	DataByte 1
000h	1 (*Reset_Node*)	*NodeID* or 0 (all nodes on the bus)

Again, there is no reply to this message.

Note that at power-up it is the Bootloader that becomes active first; it reports its presence by sending the following Emergency message (see also section 5):

Bootloader → Host

COB-ID	Byte 0-1	Byte 2	Byte 3-7
080h + *NodeID*	Emergency Error Code (00h 50h)	Error Register (Object 1001h) (80h)	Manufacturer specific error field (FEh 01h 28h ZZh 00h) (ZZh = MCUCSR)

(*MCUCSR* = MCU Control and Status Register; for details see the ATmega128 datasheet).

After about 4 s the Bootloader automatically jumps to the application. The Bootloader jumps immediately to the application, if it receives an NMT *Reset-Node* message, as shown above.

3.3 Node Guarding and Life Guarding

Node Guarding in CANopen is a mechanism whereby an *NMT-master* checks the state of other nodes on the bus, at regular intervals. It can do this in one of two different ways:

1. The master sends a Remote Transmission Request (RTR) for the Node Guard message, to each node on the bus, in turn; a node that receives the RTR, sends the Node Guard message, which contains one data byte indicating the (CANopen) state of the node, as well as a toggle bit. If a node does not reply the master should signal this to the higher-level software and/or take appropriate action.
   The RTR for the Node Guard message looks like this (a Remote Frame, so the CAN-message has no data bytes):

   Host (NMT-Master) $\rightarrow$ ELMB (NMT-Slave)

COB-ID
700h + *NodeID*

   The reply Node Guard message from a node looks like this:

   ELMB (NMT-Slave) $\rightarrow$ Host (NMT-Master)

COB-ID	DataByte 0
700h + *NodeID*	bit 7: *toggle bit*, bit 6-0: *state*

2. Each node on the bus sends a Heartbeat message at regular intervals; typically, the NMT-master monitors these messages and keeps a time-out period for each node. The master detects nodes that stop sending their Heartbeat messages and should signal this to the higher-level software and/or take appropriate action.
   A Heartbeat message looks like this:

   ELMB (Heartbeat producer) $\rightarrow$ Consumer(s) (e.g. NMT-Master)

COB-ID	DataByte 0
700h + *NodeID*	*state*

State is one of these CANopen states: 0 (*Initializing*), 4 (*Stopped*), 5 (*Operational*) or 127 (*Pre-operational*). Note that this makes the *Boot-up* message the first Heartbeat message after a node reset (see previous section).

According to the CANopen standard, a node is not allowed to support both Node Guarding and Heartbeat protocols at the same time. The framework application supports both methods of Node Guarding (but indeed not at the same time), i.e. it can send the Node Guard message or it can send the Heartbeat message with an interval, which is configurable in *OD* index 1017h.

Life Guarding in CANopen is a mechanism whereby a node checks the aliveness of the host or master, by applying a time-out on messages received. CANopen defines that the message to time-out is the RTR for the Node Guard message, sent by the NMT-master; however, the framework application resets its Life Guarding timer at each properly received message addressed to it.

Life Guarding is controlled through *OD* objects 100Ch and 100Dh. In the framework application the Life Guarding time-out can be set between 1 and 255 seconds, by setting *OD* index 100Dh to the corresponding value, or can be switched off, by setting *OD* index 100Dh to zero.

If a Life Guarding time-out occurs, the node should take whatever appropriate action. The framework application resets and reinitializes the CAN-controller, and (tries to) resume(s) normal operation, after sending an Emergency message (see section 5).

3.4 Accessing the Object Dictionary using SDO Messages

At any time after initialisation (except when the node is in *Stopped* state) ***SDO*** messages can used to read from and write to the Object Dictionary. The Object Dictionary of the framework is listed in the tables in section 4. The framework supports Expedited Transfer only, i.e. data items read or written must have a size of 4 bytes or less).

3.5 Data Read-out using PDO Messages

When the ELMB application is in Operational mode the most efficient way of communicating application data is by means of ***PDO*** messages, which carry very little overhead and require no confirmation from the receiving side.

The framework supports 4 **RPDO**s (ELMB is receiver of the data) and 4 **TPDO**s (ELMB is the sender of the data). The framework receives the RPDOs and sends the TPDOs (depending on the trigger and on the PDO's *transmission mode*), but the data is dummy and is not used to set any hardware (in the case of an RPDO) or is obtained from any hardware (in the case of a TPDO). This needs to be filled in further by the framework user.

3.6 Storing Parameters and Settings in Non-Volatile Memory

Parameters and settings can be stored permanently onboard in non-volatile memory (EEPROM) by writing string "save" to *OD* index 1010h. The *CANopen* ***SDO*** mechanism is used to do this:

Host → ELMB

COB-ID	**DataByte**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
600h + *NodeID*	23h	10h	10h	*subindex*	73h ('s')	61h ('a')	76h ('v')	65h ('e')

with *OD* index 1010h in byte 1+2 and *subindex* in byte 3 with *subindex*:
= 1: store all parameters (as listed for *subindex* 2 and 3).
= 2: store communication parameters (concerning PDO and Guarding).
= 3: store application parameters (concerning ADC, DAC and Digital I/O).
(check out the Object Dictionary tables in section 4 to find out which parameters are stored).

If the store-operation succeeded the application ends the following reply:

ELMB → Host

COB-ID	**DataByte**						
	0	**1**	**2**	**3**	**4**	**5**	**6-7**
580h + *NodeID*	60h	10h	10h	*subindex*	–	–	–

If the store-operation did NOT succeed the application sends the following reply (SDO *Abort Domain Transfer*, error reason: 'hardware fault' (for details see [1])):

ELMB → Host

COB-ID	**DataByte**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
580h + *NodeID*	80h	10h	10h	*subindex*	0	0	6 (Error Code)	6 (Error Class)

Parameters can be reset to their default values (by invalidating the corresponding contents of the EEPROM) by writing to *OD* index 1011h, using this time the string "load" (6Ch, 6Fh, 61h, 64h) in bytes 4 to 7 of the ***SDO***. Note that the default values take effect only after a subsequent reset of the node. Default values are listed in the *OD* tables in section.4

The tables with the Object Dictionary in section 4 show the settings stored in EEPROM as marked by an asterisk (*).

4 Object Dictionary

The Object Dictionary (*OD*) of the **ELMBfw v2.0** framework application is listed in the tables on the next pages. Objects can be read or written using CANopen SDO messages.

The values of objects marked with '*' in the *Index* column are stored in EEPROM for permanent non-volatile storage, on request (as described in section 3.6). They are retrieved from EEPROM at each reset or power-up and used to initialize the application.

Objects in the tables with a shaded (yellow) background (may) require changes and additions by a framework user.

Objects marked "EXPERT ONLY" are in principle 'for experts only', but it is safe to read any such Object if marked as *Readable* (R). If for any reason the data stored in these Objects is lost (and you need it for your application), please contact the ATLAS DCS team.

Communication Profile Area

Index (hex)	Sub Index	Description	Data/ Object	Attr	Default	Comment
1000	-	Device type	U32	RO	00000000h	Meaning: no *CANopen* device profile supported; mandatory CANopen object
1001	-	Error register	U8	RO	0	
1002	-	Manufacturer status register	U32	RO	0	[1] (see footnote)
1008	-	Manufacturer device name	String	RO	"ELMB"	= Embedded Local Monitor Board
1009	-	Manufacturer hw version	String	RO	"el40"	= ELMB V4
100A	-	Manufacturer software version	String	RO	"FW20"	***ELMBfw*** application version 2.0
100C	-	Guard time [ms]	U16	RO	1000	= 1 second
100D *	-	Life time factor	U8	RW	0	lifeguarding timeout in seconds; 0 → no lifeguarding timeout
1010		Store parameters	Array			Save stuff in onboard EEPROM
	0	Highest index supported	U8	RO	3	
	1	Save all parameters	U32	RW	1	Read: 1; Write "save": store all
	2	Save communication parameters	U32	RW	1	Read: 1; Write "save": store PDO par's, Life time factor, …
	3	Save application par's	U32	RW	1	Read: 1; Write "save": store application configuration
1011		Restore default parameters	Array			Invalidate stuff in onboard EEPROM; use defaults afterwards
	0	Highest index supported	U8	RO	3	
	1	Restore all parameters	U32	RW	1	Read: 1; Write "load": invalidate all parameters stored
	2	Restore communication parameters	U32	RW	1	Read: 1; Write "load": invalidate stored PDO par's, etc.
	3	Restore application par's	U32	RW	1	Read: 1; Write "load": invalidate stored application config
1017 *	-	Producer Heartbeat Time [1 s]	U16	RW	0	In units of seconds (but <=255 !), (NB: should be in ms according to CANopen!); 0 → Heartbeat is disabled
1018		Identity	Record			Mandatory CANopen object
	0	Number of entries	1..4	RO	1	
	1	Vendor ID	U32	RO	12345678h	*to be ordered from CiA*

[1] Manufacturer Status Register contains the application-specific error bits.

Communication Profile Area ***(continued…)***

Index (hex)	Sub Index	Description	Data/ Object	Attr	Default	Comment
1400		1st Receive PDO par's	Record			Data type = PDOCommPar
	0	Number of entries	U8	RO	5	
	1	COB-ID used by PDO	U32	RO	200h + *NodeID*	According to CANopen Prede-fined Connection Set
	2	Transmission type	U8	RO	255	Only 255 allowed
	3,5	*Not used*		RO	0	
1401		2nd Receive PDO par's	Record			(as above, with COB-ID 300h)
1402		3rd Receive PDO par's	Record			(as above, with COB-ID 400h)
1403		4th Receive PDO par's	Record			(as above, with COB-ID 500h)
1600		1st Receive PDO mapping	Record			Data type = PDOMapping
	0	Number of entries	U8	RO	2	
	1	Digital outputs 1-8 *(Example: object referred to does not exist)*	U32	RO	62000108	OD index 6200, sub-index 1: Outputs 1-8, size = 8 bits
	2	Digital outputs 9-16 *(Example: object referred to does not exist)*	U32	RO	62000208	OD index 6200, sub-index 2: Outputs 9-16, size = 8 bits
1601		2nd Receive PDO mapping	Record			
1602		3rd Receive PDO mapping	Record			
1603		4th Receive PDO mapping	Record			
1800		1st Transmit PDO par's	Record			Data type = PDOCommPar
	0	Number of entries	U8	RO	5	
	1	COB-ID used by PDO	U32	RO	180h + *NodeID*	According to CANopen Prede-fined Connection Set
*	2	Transmission type	U8	RW	1	Only 1 and 255 allowed
	3	Inhibit time [100 µs]	U16	RO	0	*not used*
*	5	Event timer [1 s]	U16	RW	0	In units of seconds (NB: should be in ms according to CANopen!); active if >0 and transmission-type = 255
1801		2nd Transmit PDO par's	Record			(as above, with COB-ID 280h)
1802		3rd Transmit PDO par's	Record			(as above, with COB-ID 380h)
1803		4th Transmit PDO par's	Record			(as above, with COB-ID 480h)
1A00		1st Transmit PDO mapping	Record			Data type = PDOMapping
	0	Number of entries	U8	RO	2	
	1	Digital inputs 1-8 *(Example: object referred to does not exist)*	U32	RO	60000108	OD index 6000, sub-index 1: Inputs 1-8, size = 8 bits
	2	Digital inputs 9-16 *(Example: object referred to does not exist)*	U32	RO	60000208	OD index 6000, sub-index 2: Inputs 9-16, size = 8 bits
1A01		2nd Transmit PDO mapping	Record			
1A02		3rd Transmit PDO mapping	Record			
1A03		4th Transmit PDO mapping	Record			

Manufacturer-Specific Profile Area *(continued…)*						
Index (hex)	**Sub Index**	**Name**	**Data/ Object**	**Attr**	**Default**	**Comment**
2000		Any application-specific settings	Array			This Object has been added as an example only: see source code for handling reading and writing to the Object Dictionary
	0	Number of entries	U8	RO	1	
	1	Number of channels	U16	RW	4	
	…	…	…	…		…
	…	…	…	…		In the Index range 2000-5FFF a user can add any Objects needed for his application
…	…	…	…	…		…

Manufacturer-Specific Profile Area *(continued…)*						
Index (hex)	**Sub Index**	**Name**	**Data/ Object**	**Attr**	**Default**	**Comment**
2A00		ADC range calibration	Array		**EXPERT ONLY**	For now triggers a 'pure' self-calibration procedure only [1]
	0	Number of entries	U8	RO	6	
	1	Calibrate 25 mV	U32	WO		Write any value…
	2	Calibrate 55 mV	U32	WO		Write any value…
	3	Calibrate 100 mV	U32	WO		Write any value…
	4	Calibrate 1 V	U32	WO		Write any value…
	5	Calibrate 2.5 V	U32	WO		Write any value…
	6	Calibrate 5 V	U32	WO		Write any value…
2B00		ADC calibration parameters 25 mV	Array			Calibration constants (always stored in EEPROM); enable by first writing to 2D00
	0	Number of entries	U8	RO	4	
*	1	Gain Factor phys. chan. 1	U32	RW		actual gain factor * 1000000
*	2	Gain Factor phys. chan. 2	U32	RW		actual gain factor * 1000000
*	3	Gain Factor phys. chan. 3	U32	RW		actual gain factor * 1000000
*	4	Gain Factor phys. chan. 4	U32	RW		actual gain factor * 1000000
2B01		ADC calibration parameters 55 mV	Array			Calibration constants (as above)
2B02		ADC calibration parameters 100 mV	Array			"
2B03		ADC calibration parameters 1 V	Array			"
2B04		ADC calibration parameters 2.5 V	Array			"
2B05		ADC calibration parameters 5 V	Array			"
2C00	-	Erase ADC calibration parameters 25 mV	U8	WO	**EXPERT ONLY**	Write EEh to erase; enable by first writing to 2D00
2C01	-	Erase ADC calibration parameters 55 mV	U8	WO	**EXPERT ONLY**	"
2C02	-	Erase ADC calibration parameters 100 mV	U8	WO	**EXPERT ONLY**	"
2C03	-	Erase ADC calibration parameters 1 V	U8	WO	**EXPERT ONLY**	"
2C04	-	Erase ADC calibration parameters 2.5 V	U8	WO	**EXPERT ONLY**	"
2C05	-	Erase ADC calibration parameters 5 V	U8	WO	**EXPERT ONLY**	"
2D00	-	Enable calibration parameter write/erase operation	U8	WO	**EXPERT ONLY**	Writing 0xA5 enables one write or erase operation to any of the Objects 2B00 to 2B05 or 2C00 to 2C05.

[1] In other words: reset the ADC and do a 'self-calibration', i.e. do **NOT** apply the gain factors ('calibration constants'), which might have been downloaded to EEPROM already. This type of ADC initialisation is essential when <u>re</u>calibrating the voltage range in question.

Manufacturer-Specific Profile Area

Index (hex)	**Sub Index**	**Description**	**Data/ Object**	**Attr**	**Default**	**Comment**
3000		Calculate 16-bit CRC	Array			
	0	Number of entries	U8	RO	2	
	1	CRC of Master's program code in FLASH	U16	RO	0	SDO reply unequal to zero means there is a checksum error; absence of CRC results in SDO *Abort* with *Error Code* 1; error while accessing FLASH results in SDO *Abort* with *Error Code* 6.
	2	CRC of Slave's program code in FLASH	U16	RO	0	*Idem (only forELMB103 with ATmega103 + slave processor)*
	3	Get (Master code) CRC	U16	RO		Return CRC from flash
3100	-	ELMB Serial Number	U32	RW		Number or 4-byte string uniquely identifying an ELMB, given during production.
3101	-	Enable ELMB Serial Number write operation	U8	WO	**EXPERT ONLY**	Writing 5Ah enables one write operation on the Serial Number (Object 3100).
3200		CAN-controller settings	Array			
	0	Number of entries	U8	RO	3	
*	1	Disable Remote Frames	U8	RW	0	[1]
*	2	Enable auto-start	U8	RW	0	If =1 go to *Operational* at startup
*	3	Bus-off max retry counter	U8	RW	5	A counter is decremented every 1 s and incremented every time bus-off occurs, but if it reaches this maximum value the node abandons regaining CAN-bus access at bus-off; if value=255 the node retries indefinitely.
5C00	-	Compile Options	U32	RO		Bitmask denoting which compile options were used when the application was generated (see table below for details)
5DFF		ELMB Tests	Array		**EXPERT ONLY**	*For use in ATLAS DCS production and test stand only*
	0	Number of test objects	U8	RO	1	
	1	Test of I/O-pins	U32	RO		see description in another doc

[1] Due to the way the ELMB's CAN-controller handles Remote Frames, it is recommended to disable Remote Frames permanently if not needed (for PDO read-out). A special provision in the software has been made to ensure that the Node Guard Remote Frame is still handled properly.

Manufacturer-Specific Profile Area						
Index (hex)	**Sub Index**	**Description**	**Data/ Object**	**Attr**	**Default**	**Comment**
5E00	-	Transfer control to Boot-loader (or ELMB103 Slave-processor)	U8	WO		It takes a few seconds (ca. 4 s) before the *Slave* processor takes control (ELMB103) ; transfer to the Bootloader is immediate (ELMB128)
5F50		Download ELMB103 Slave Program Data	Array			For reprogramming the ELMB AT90S2313 *Slave* processor
	0	Number of supported pro-grams	U8	RO	-	*not implemented*
	1	Program number 1	U32	WO		Data bytes contain one read / write instruction (for a single byte) for microcontroller's FLASH or EEPROM memory, according to the AT90S2313 Serial Programming Instruction Set (see AT90S2313 datasheet)

Object 5C00: Compile Options		
Bit	***Compile Option***	***Comment***
0	–	–
1	–	–
2	–	–
3	–	–
4	–	–
5	_7BIT_NODEID_	only DIP-switch 1 used for CAN baudrate (125 or 250 kbaud); other 7 switches used for setting Node-ID: 1-127 (when this option is not set a 6-bit Node-ID is used and 2 bits are used for selecting a baudrate) NB: do not use, as it clashes with the DIP-switch usage by the Bootloader.
6	–	–
7	_ELMB103_	the ELMB is an ELMB103 type (with ATmega103 processor); by default an ELMB128 (with ATmega128 processor) is assumed
8	_VARS_IN_EEPROM_	store/retrieve working copies of configuration parameters in/from EEPROM; this increases the radiation-tolerance of the firmware
9	–	–
10	_INCLUDE_TESTS_	include an OD object through which (board) tests can be executed; may be useful when an ELMB is removed for repairs
11	–	–
12	_CAN_REFRESH_	refresh CAN-controller descriptor register (at each buffer write/read); this increases the radiation-tolerance of the firmware
13	_2313_SLAVE_PRESENT_	there is (probably) a Slave processor (usually when using an ELMB103, so in combination with compile option *_ELMB103_*); this includes the code that deals with the Slave processor

Table 5. Optional compiler macro defines.

5 Emergency Objects

Emergency messages are triggered by the occurrence of an internal (fatal) error situation. An emergency CAN-message has the following general syntax:

ELMB → Host

COB-ID	Byte 0-1	Byte 2	Byte 3-7
080h + *NodeID*	Emergency Error Code	Error Register (Object 1001h)	Manufacturer specific error field

The following Emergency messages can be generated by the **ELMBfw** framework application:

Error Description	Emergency Error Code (byte 0-1)	Manufacturer-Specific Error Field (byte 3-7)
CAN communication	8100h	Byte 3: 81C91 Interrupt Register content Byte 4: 81C91 Mode/Status Register content Byte 5: error counter
CAN buffer overrun	8110h	CAN message buffer in RAM full: at least 1 message was lost
Life Guarding time-out	8130h	(CAN-controller has been reinitialized)
RPDO: too few bytes	8210h	Byte 3: minimum DLC (Data Length Code) required
Slave processor not responding	5000h	Byte 3: 20h
CRC error	5000h	Byte 3: 30h Byte 4: 1 (Master FLASH), 2 (Slave FLASH)
EEPROM: write error	5000h	Byte 3: 41h Byte 4: Parameter block index [1] Byte 5: 0 : writing block info > 0: size of parameter block to write
EEPROM: read error	5000h	Byte 3: 42h Byte 4: Parameter block index [1] Byte 5: Error id (1=CRC, 2=length, 4=infoblock)
Irregular reset	5000h	Byte 3: F0h Byte 4: microcontroller MCUCSR register contents [2]
No Bootloader	5000h	Byte 3: F1h

…table continues on the next page…

[1] **0**: TPDO communication parameters, **1**: RPDO communication parameters, **2**: Guarding parameters, **3**: CAN configuration parameters, **4**: Application configuration, **FEh**: Calibration constant(s), **FFh**: ELMB Serial Number.

[2] ATmega128 *MCUCSR* register bits: **01h**: Power-On Reset, **02h**: External Reset, **04h**: Brown-Out Reset, **08h**: Watchdog Reset, **10h**: JTAG Reset, **80h**: JTAG Interface Disable

Error Description	**Emergency Error Code** (byte 0-1)	**Manufacturer-Specific Error Field** (byte 3-7)
Bootloader is in control [1]	5000h	Byte 3: FEh Byte 4: 01h Byte 5: 28h Byte 6: microcontroller MCUCSR register contents [2] Byte 7: 00h
Bootloader cannot jump to application: invalid [1]	6000h	Byte 3: FEh Byte 4: AAh Byte 5: AAh

Byte 2 of the Emergency message contains the value of the socalled *Error Register* (Object Dictionary index 1001h, a mandatory CANopen object). One or more bits of the 8-bit Error Register can be set to 1, depending on the node's history of errors since the last reset. The table below gives a description of the meaning of the different bits.

Error Register (Object 1001h) bits	
Bit	**Error type**
0	Generic
1	Current
2	voltage
3	Temperature
4	Communication
5	device profile specific
6	*reserved (=0)*
7	manufacturer specific

References

[1] CAN-in-Automation e.V.,
CANopen, Application Layer and Communication Profile,
CiA DS-301, Version 4.0, 16 June 1999.

[2] H.Boterenbrood,
CANopen Application Software for the ELMB128,
Version 2.1, NIKHEF, Amsterdam, 2 March 2004.
(http://www.nikhef.nl/pub/departments/ct/po/html/ELMB/ELMB21.pdf

[3] **ELMB software resources webpage:**
http://www.nikhef.nl/pub/departments/ct/po/html/ELMB/ELMBresources.html

[1] This Emergency message is generated by the Bootloader.
[2] ATmega128 *MCUCSR* register bits: **01h**: Power-On Reset, **02h**: External Reset, **04h**: Brown-Out Reset, **08h**: Watchdog Reset, **10h**: JTAG Reset, **80h**: JTAG Interface Disable