The emBODY for DSPIC33

This document describes the emBODY for DSPIC33.

Approval History

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Author | | Date | Approved | | Date |
| 0.1 | Accame | IIT - iCub Facility |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 0.1 | 21 Mar 12 | M. Accame | First WIP version. |
| 1.0 | 07 Jan 13 | M. Accame | Made it version 1.0 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 1](#_Toc396314132)

[2 Partitioning of the MPU 2](#_Toc396314133)

[2.1 The system memory 2](#_Toc396314134)

[2.2 The bootloader 2](#_Toc396314135)

[2.3 The Application 3](#_Toc396314136)

[2.3.1 A special application: the updtBootloader 3](#_Toc396314137)

[2.3.2 Using the updtBootloader for upgrading from bootloader version 1.x 3](#_Toc396314138)

[2.4 The Storage 4](#_Toc396314139)

[3 Overview of available libraries 6](#_Toc396314140)

[3.1 Libraries in embedded environment 6](#_Toc396314141)

[3.1.1 The eEsharedStorage 6](#_Toc396314142)

[3.1.2 The eEbasicStorage 6](#_Toc396314143)

[3.1.3 The eEsharedInfo 6](#_Toc396314144)

[3.1.4 The eEpermData 6](#_Toc396314145)

[3.2 The HAL 6](#_Toc396314146)

[3.3 The embOBJ 6](#_Toc396314147)

[4 Programming paradigms on DSPIC 7](#_Toc396314148)

[4.1 The basic application 7](#_Toc396314149)

[4.2 The HAL application 7](#_Toc396314150)

[4.3 The embOBJ application 8](#_Toc396314151)

[5 An example of application: the bootloader 9](#_Toc396314152)

# Introduction

This document describes the emBODY for DSPIC33 used in boards such as the 2FOC and the 6SG.

The emBODY is a set of SW conventions and of libraries which allow developing applications in a flexible way: from the basic forever loop to a more structured object-oriented manner.

In the following there is described how the MPU is partitioned to contain three main sections: the bootloader process, the application process, and the storage for use of the former two.

It is also shown what are the SW modules used in the SW of the bootloader, used for manipulation of the storage section, and also those which can be used for the application.

Finally, some possible programming paradigms are presented to accommodate the diverse uses of a potentially wide range of users.

Partitioning of the MPU

The flash of the DSPIC33 is allocated as it follows.

The sizes of the sections and their positions are contained in the file eEmemorymap.h.

MAPPING OF FLASH ON DSPIC33 USING EMBODY

SYSTEM MEMORY

APPLICATION

BOOTLOADER

STORAGE

0x00000

0x00200

0x0D000

0x14C00

0x15800

Keeps jump address at bootstrap (0xD000), and the addresses of the ISRs

Keeps the application launched by the bootloader.

This code is executed at bootstrap. The bootloader initializes CAN bus, waits FW upgrade commands and if not received it launches the application. If received it starts updating mode and loads in FLASH a new application.

Keeps CRC16-validated data for the use of the bootloader and of the application.

**Figure 1**: Partitioning of the FLASH of the DSPIC33.

## The system memory

It contains the address at which the system jumps after a reset and also the address of the ISRs. It is manipulated by the bootloader.

Normally, the jump address is the start of the bootloader process.

## The bootloader

It contains the code executed just after a reset.

The bootloader has two functionalities: it normally launches the application, but it can also enter in update mode and burn in FLASH a new application.

At startup, the bootloader initialises the CAN bus and waits for a small amount of time before launching the application. If there is not a valid application the system resets and the bootloader starts again.

But if during the initial wait time, the bootloader retrieves a proper CAN message, then it enters in update state.

At startup the bootloader performs another important task: it make sure that some sections of the storage are properly initialised with information such as a partition table of the system, name and version of the HW board, FW version of the bootloader, CAN address etc. Further details in the storage section.

## The Application

The application contains code about what the board really needs to do. Further details on how it is possible to write SW for the application in a later section.

### A special application: the updtBootloader

The bootloader can load the updtBootloader application which enables the canLOADER to load a new bootloader in the system.

The updtBootloader does not have dependencies from the system thus is a flexible instrument for changing not only the bootloader code, but also the mapping in FLASH of the system and the format of the storage.

However, the main interest is that it is backwards compatible with the old bootloader which relies on a different memory mapping.

### Using the updtBootloader for upgrading from bootloader version 1.x

The upgrade from the old bootloader (version 1.x) towards the new one is described in the following for the case of the 2FOC board.

The 2FOC board has two MPUs and exits the test procedure with CAN ID 12 and 13. In the whole procedure, DON’T SWITCH POWER OFF unless explicitly written.

1. It is important to connect to the CAN bus only one 2FOC board. The canLOADER will show two MPUs with addresses 12 and 13 and FW version 1.1.0.
2. Tick ID 12 and load the updtBootloader. At the end of the operation, the canLOADER will show ID 10 and FW version 100.1.0 (or later). DON’T EVER SWITCH OFF in here.
3. Tick ID 10 and load the bootloader. At the end of the operation the canLOADER will show ID 14 and FW version 2.1.0 (or later).
4. Tick ID 14 and change its address into 12 again.
5. Now remove as soon as possible the updtBootloader by loading the 2FOC.hex application. You MUST be sure to tick ID 12 (and NOT ID 13). At the end of the operation, the canLOADER will show ID 12 and the FW version of the target application.
6. Now you can repeat points from 2 until 5 using ID 13.

It any failure in this procedure will require the use of JTAG programming.

The risks in here are when:

1. We have the updtBootloader written in the memory of the MPU but still the old bootloader. If a restart happens in such a case, then after reset the MPU jumps at the address of the new bootloader which is not present. That is why we MUST NOT power off at point 2.
2. We have the old bootloader and we use it to write the new 2FOC.hex application. The operation is likely to be successful, but if a restart happens, after reset the MPU jumps at the address of the new bootloader which is not present. That is why at point 5 you must be sure to load the application at the right ID.

## The Storage

The storage contains information protected by a CRC16 and used by the system. It is divided in three groups: read only information shared by the bootloader and the application, read write shared information, and read write data reserved for the application.

The first group contains information which is written only once by the bootloader at its first go and is only read after this moment: the partition table of the system, the version of the HW board, and the version of the bootloader.

The second group contains information which is typically read but which can be written several times: the device info (so far with inside only the CAN address), the version of the application, and inter-process communication data which is used to exchange messages between the bootloader and the application across a reset.

The data structure stored inside is described in eEsharedInfo.h and in eEcommon.h.

The content of the storage is manipulated by the library eEsharedStorage which manages operations on the internal FLASH and on the protection CRC.

A ultra-light library, the eEbasicStorage, is also available to perform only CAN address retrieval.

THE STORAGE SECTION ON DSPIC33 USING EMBODY

STRG-SHARED-RO

STRG-SHARED-RW

STRG-APPLICATION

1KB

1KB

1KB

Bootloader Info

Board Info

Partition Table

UNUSED

128B

128B

128B

Interprocess Communication

Application Info

Device Info

UNUSED

128B

128B

128B

Reserved for the APPLICATION

**Figure 2**: Details of the storage section.

Overview of available libraries

On DSPIC33 it is possible to use a number of utility libraries which are specific of the architecture, some which have a set of APIs shared with other MPUs in the emBODY but the implementation of which is specific of the DSPIC33, and finally others which are truly platform independent and thus are shared by every MPU in the emBODY.

The first libraries are those in the embedded environment section. The ones which shares the API are limited to a hardware abstraction layer, and the truly platform independent are the embOBJ.

## Libraries in embedded environment

They are the following.

### The eEsharedStorage

It allows an application or the bootloader to correctly access to the shared storage section. It uses functionalities of eEsharedInfo which in chain uses eEpermData and the HAL.

### The eEbasicStorage

It allows a very basic access to the storage section with functionalities so far limited to CAN address validated retrieval. It uses only the functions provided by the silicon vendor.

### The eEsharedInfo

It allows manipulation of data in storage from a lower level than eEsharedStorage. For memorisation if uses eEpermData.

### The eEpermData

It allows storage of generic data in any eeprom kind. It uses FLASH and CRC16 functionalities of HAL.

## The HAL

It shares the same API in the whole emBODY but it implements only the subset of functionalities which are allowed by the DSPIC33 and which are of help in the development of applications: ARCH-DSPIC specific, BASE, CAN, CRC, EEPROM, FLASH, GPIO, LED, SYSTEM, TIMER.

## The embOBJ

They are used in some applications, such as the 6SG, in the version called SEE (singletask execution environment).

Programming paradigms on DSPIC

On emBODY it is possible to write applications which use diverse programming styles and approaches, from the minimal forever loop with no HAL at all to the most advanced EOtheFOOP available in the embOBJ.

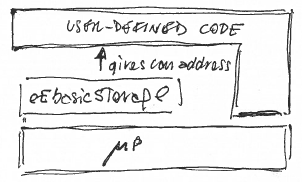
Here are some examples of frameworks given in order of decreasing complexity of the code to write inside them ;-).

## The basic application

It is found in the “appl-basic.mcw” project and contains just:

* a linker script coherent with the used eEmemorymap.h,
* the eEbasicStorage.c file to be able to read the CAN address stored in STORAGE area,
* a main.c file with the main() function with retrieval of the CAN address.

Any other code is to be written. The project is with minimal constraints and is good for importing other projects not originally developed under emBODY. As an example see the 2foc-icub project.



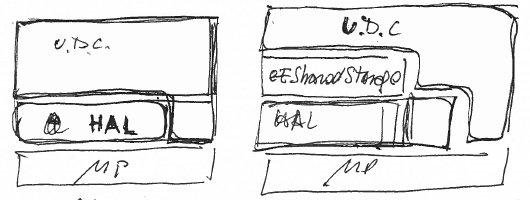
## The HAL application

It is found in the “appl-hal.mcw” project and contains:

* a linker script coherent with the used eEmemorymap.h,
* the HAL source files with proper configuration and storage files,
* the eEsharedStorage.c file plus other subordinates which allows to fully interface to STORAGE area,
* a main.c file with the main() function which calls an init() function and then it enters in a forever loop which repeats a run() function,
* other three files with the definition of the init() and of the run() function.

The applications is able to write in STORAGE its version and information, to initialise the HAL with CAN bus, EEPROM, CRC16, LED, to retrieve message and to send replies back.

The files to be personalised by the user are “mainly” those in the app folder. The user takes advantage of consolidated APIs and of ready and proven libraries.



## The embOBJ application

It is found in the “appl-embobj.mcw” project and contains:

* a linker script coherent with the used eEmemorymap.h,
* the HAL source files with proper configuration and storage files,
* the eEsharedStorage.c file plus other subordinates which allows to fully interface to STORAGE area (TO BE ADDED because IT IS MISSING SO FAR: maybe use an object EOtheDSPICenvironment),
* a main.c file with the main() function which initialises and runs the EOStheSystem object of the embOBJ,
* other files with the examples of use of some embOBJs.

The applications can exploits advanced SW services such as timers, events, messages, callbacks, state machines, containers, , etc. all available with the same APIs across the entire embedded platform (with objects or types EOtimer, eOevent\_t, eOmessage\_t, eOcallback\_t, EOsm, EOdeque, EOvector, EOlist, etc.).

These services are offered by the singleton EOStheSystem via the singleton EOtheFOOP, which runs a FOrever loOP with checks of some shared variables which are modified by calls to the objects or by ISRs, which rely on protection vs concurrent access using HAL calls.

At the very end the MPU executes the same C code as other consolidated approaches do, but under the cosmetics of high-level APIs. See “An Embedded Software Primer, David E. Simon, Addison Wesley, 1999”.

An example of program: the bootloader

The bootloader is built upon the HAL application of the previous section and uses the eEsharedStorage to manipulate the storage. It also integrates a state machine to take tracks of the complex actions it must perform, a parser of the received CAN frame, and a manager for the memory,

The used libraries and the execution state machine are reported in the following figures.

SW PARTS USED IN THE BOOTLOADER

HAL

P’s registers or libraries by MicroChip

eEsharedStorage

State Machine

CAN Parser

Memory Manager

User Defined Code

Ready Libraries

**Figure 3**: Libraries used by the bootloader.

BASIC STATE MACHINE OF THE BOOTLOADER

INIT

LOADER

UPDATER

go2updater

programming-is-finished

timeout-is-expired

**Figure 4**: Basic state machine of the bootloader.

SEQUENCE DIAGRAM IN LOADER MODE

HAL

State Machine

(1) Synchronizes info

eEsharedStorage

(4) if no orders to switch to updater mode it jumps to application

CAN Parser

(2) checks for incoming CAN frames

(3) if any, it parses them

**Figure 5**: Sequence diagram in Loader mode.

SEQUENCE DIAGRAM UPDATER MODE

HAL

State Machine

(4b) if needed it sends frames back

(4a) if any applic data it writes it in FLASH

CAN Parser

(1) retrieves an incoming CAN frame

(2) parses it

Memory Manager

P’s registers or libraries by MicroChip

(4) evals opcode

(4c) if finished it jumps to application

(3) gives back opcode and data

**Figure 6**: Updater mode.