TFT Cluster Software Architecture Specification

Contents

[1 Overview 6](#_Toc429646922)

[2 The Application and Platform services layer 6](#_Toc429646923)

[2.1 Platform services 7](#_Toc429646924)

[2.1.1 Resource Manager 7](#_Toc429646925)

[2.1.2 Display Manager 7](#_Toc429646926)

[2.1.3 Label Manager 7](#_Toc429646927)

[2.1.4 Text Renderer 7](#_Toc429646928)

[2.1.5 Check message handler 7](#_Toc429646929)

[2.2 Function Blocks 8](#_Toc429646930)

[2.2.1 Function Block file layout 8](#_Toc429646931)

[2.2.2 Function Definition File (FDF) 8](#_Toc429646932)

[2.2.3 Source and header files 8](#_Toc429646933)

[2.2.4 Configuration data 8](#_Toc429646934)

[2.3 Function Library 9](#_Toc429646935)

[2.4 Interface description 9](#_Toc429646936)

[3 The Platform component layer 9](#_Toc429646937)

[3.1 GCL 9](#_Toc429646938)

[3.1.1 Signal mapping 10](#_Toc429646939)

[3.1.2 The Signal Mapping File 10](#_Toc429646940)

[3.1.3 Signal interface 10](#_Toc429646941)

[3.1.4 Calibration interface 11](#_Toc429646942)

[3.1.5 Diagnostics interface 11](#_Toc429646943)

[3.2 Scheduler 11](#_Toc429646944)

[3.2.1 Scheduler and BSP software integration 11](#_Toc429646945)

[3.2.2 Task software interface 11](#_Toc429646946)

[3.2.3 Scheduler configuration 12](#_Toc429646947)

[3.3 Function Block Specific GCL header 12](#_Toc429646948)

[3.3.1 Input signals 12](#_Toc429646949)

[3.3.2 Output signals 12](#_Toc429646950)

[3.4 COM stack 13](#_Toc429646951)

[3.4.1 COM 13](#_Toc429646952)

[3.4.2 PDU Router 13](#_Toc429646953)

[3.4.3 CAN-IF 13](#_Toc429646954)

[3.5 Calibration manager 13](#_Toc429646955)

[3.5.1 Cal retrieval interface 14](#_Toc429646956)

[3.5.2 Cal write interface 14](#_Toc429646957)

[3.6 Diagnostic interface 14](#_Toc429646958)

[3.6.1 Task interface 14](#_Toc429646959)

[3.6.2 Diagnostic event interface 14](#_Toc429646960)

[3.6.3 Calibration interface 15](#_Toc429646961)

[3.6.4 Communication interface 15](#_Toc429646962)

[3.7 Mode Manager 16](#_Toc429646963)

[3.7.1 State chart 17](#_Toc429646964)

[3.7.2 Propagation of ECU Mode: 17](#_Toc429646965)

[3.7.3 General transition behavior 18](#_Toc429646966)

[3.7.4 Mode descriptions 19](#_Toc429646967)

[3.7.5 Software Interfaces: 21](#_Toc429646968)

[4 The Platform Layer 21](#_Toc429646969)

[4.1 HAL layer 22](#_Toc429646970)

[4.2 BSP Interface 22](#_Toc429646971)

[4.3 Target BSP 22](#_Toc429646972)

[4.4 PC-BSP (Simulator) 22](#_Toc429646973)

[4.5 CAN Driver (Target/Simulator) 22](#_Toc429646974)

[4.5.1 CAN-Driver 23](#_Toc429646975)

[5 Bootloader 24](#_Toc429646976)

[5.1 MCU startup sequence 24](#_Toc429646977)

[5.2 Interface Bootloader and BSP 25](#_Toc429646978)

[5.2.1 Memory addresses 25](#_Toc429646979)

[5.2.2 Boot target registry 25](#_Toc429646980)

[5.2.3 BSP start function 25](#_Toc429646981)

[6 Software Structure 25](#_Toc429646982)

[6.1 Generated software structure 25](#_Toc429646983)

[6.1.1 The Application Folder 26](#_Toc429646984)

[6.1.2 The platform 26](#_Toc429646985)

[6.1.3 The MCAL folder 26](#_Toc429646986)

[6.1.4 The Makefile 26](#_Toc429646987)

[6.2 Linker considerations 26](#_Toc429646988)

[7 Glossary 27](#_Toc429646989)

Revision History

| Revision | Date | Author | Changelog |
| --- | --- | --- | --- |
| 1 | 2015-10-23 | Laan | Initial version for TFT cluster |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

s

# Overview

The IC software is divided in to three main layers; Microcontroller abstraction layer (MCAL), platform components (PC), and application & Platform services (A&PS)

The (A&PS) layer consist of the included function blocks and platform services such as menu manager, the PC layer of the components that are needed to communicate and access memory and the MCAL layers abstracts the other layers from the underlying hardware.

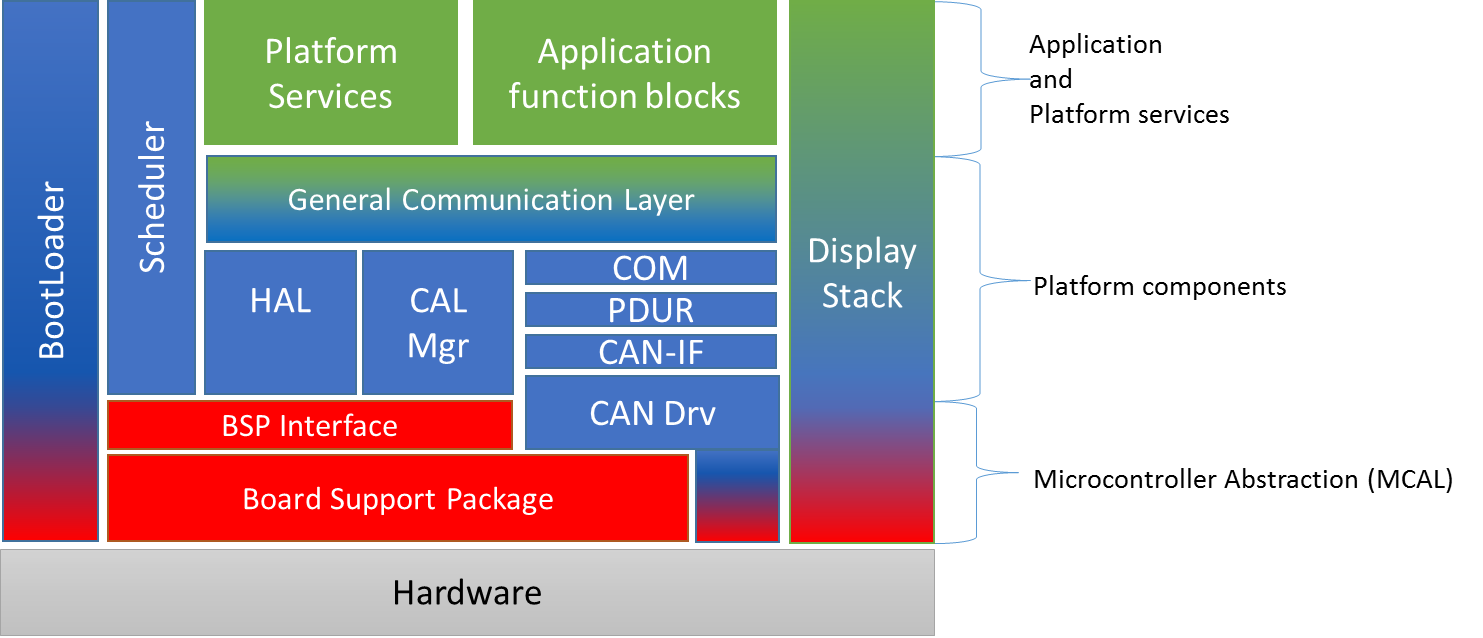


Figure 1 - Architecture overview

# The Application and Platform services layer

The A&PS layer contains of two parts: The function blocks that have been included in the project and platform services which are also included. The platform services are blocks that are generated based on configuration by the users. This includes blocks such as the menu and display manager, the checkmessage handler, label and font manager, resource manager and text renderer. The graphical components are part of the display stack which stretch from the top layers down to the microcontroller.

Communication between function blocks and to the BSP is performed via the GCL. The platform services are accessed via direct function calls into the respective block. The scheduler manages the execution of both the function blocks and the platform services.

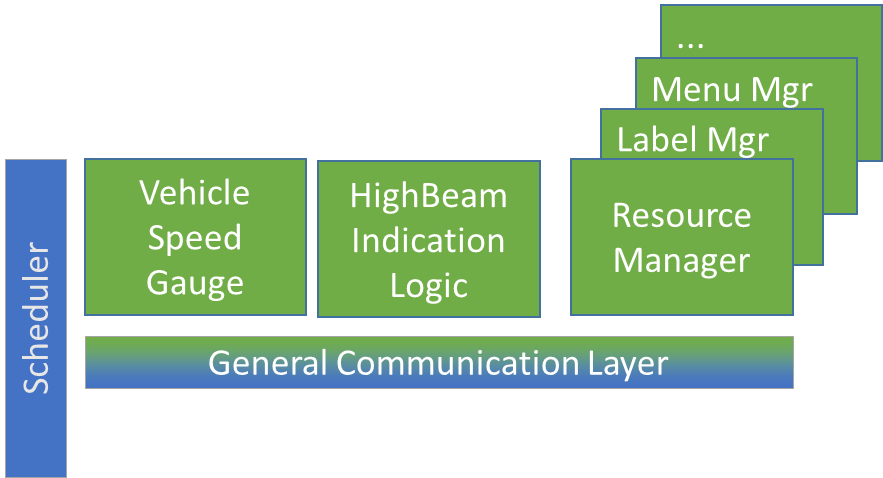


Figure 2 - The Application & PS layer

## Platform services

The platform services are special application blocks that are generated from the FDT. The blocks that are generated are used to manage core functionality of the cluster.

### Resource Manager

The resource manager manages all the resources of the cluster. At present the resources is limited to the display. Buttons are seen as an implicit resource that is owned by the resource client that owns the display.

### Check message handler

The check message handler manages all the messages that shall be shown on the display. Each check message are defined by a layout that it shall use, the text labels it shall use and a trigger function that shall be used to determine if a message shall be shown or not. The trigger signal usually comes from a logic block (trigger function). The trigger function shall set the value >0 if the messages shall be triggered.

### Display stack

The graphical components in the top layer are responsible for the interaction with the functionblocks. They manage the rendering of dynamic content that will be used by the lower layers that manage the DCU.

#### Display Manager

The display manager is responsible for managing the DCU. It provides the animations for the different areas of the display and manages the different layers of the DCU.

The display manager will utilize the layers of the DCU as well as the DMA capability of the MCU in order to produce images and animations. During the upcoming design of the display stack for the TFT cluster the display manager might be divided into subcomponents that will have different areas of responsibility.

#### Label Manager

Static text strings that are used in the display are defined by the label tool and given an ID. When specifying static text strings the ID shall be used to query the label tool for a specific text. The label tool then returns the correct translation for the specified string.

#### Text Renderer

The text renderer uses the string from the label manager to render the text on the display. The rendering uses the font specified for the string.

## Function Blocks

Function blocks consist of a function definition, in the form of an FDF, and a number of source files. The purpose of defining function blocks is to allow for reuse in multiple cluster software projects.

### Function Block file layout

The FDF shall be placed in the root folder of the function block. All source files must be placed in a “src” sub-folder, and header files in the “inc” sub-folder. Configuration parameters that is defined in the fdf-file will be generated to the “cfg” folder by the FDT tool. The hierarchy within the “src” and “inc” folders is not regulated. Any other folders or files placed in the root folder will be ignored, this leaves room for tests, documentation and other files of interest for the function block.

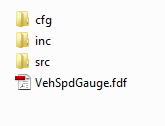


Figure 3 - Sample function block folder structure

### Function Definition File (FDF)

The FDF is an XML file which describes the name, the tasks and the signal interface for the function block. The file also defines the configuration parameter that can be assigned values in the FDT tool.

The name is used primarily for identification in the tools as well as a unique identifier in the GCL, thus the name needs to be unique in the function library.

A task consists of the name of an initialization function, the name of a runnable function (to be executed periodically), and a startup delay and a periodicity (both in milliseconds). Each task will be added to, and executed by, the scheduler according to these parameters.

The signals section describes the required and provided information by a function block. It is divided in two sub-sections, inputs and outputs. Each signal, both input and output, is described by a name and a data type. The name only has to be unique in the context of the function block, i.e. there can be a signal with the same name specified in several different function blocks.

The configuration center specifies the type, name, description of configuration parameters that are available to the user to the specific function block. If a user have defined values they will be stored in the project file.

### Source and header files

All source and header files will be copied to the output directory when running the code generator. Compilation will include all files so it is a good practice to prefix function names with the function block name to avoid redefinition of existing functions.

### Configuration data

The FDT will generate a file with the name <FunctionBlockName>\_cfg.c. The configuration parameters will be defined as const parameters with the type as defined in the FDF. The source file for the function block can use the configuration parameters by declaring them as “extern” in the source file.

## Function Library

The function library is a folder, preferably source controlled, where the function blocks are stored. Each function block has its own sub-folder containing files as defined in 2.2.2 .

## Interface description

The GCL provided the interface that the function blocks will use to communicate thru. This consists of three parts:

* Task interface
  + Initialization interface
  + Runnable interface
  + Timer access interface
* Signal interface
  + Read signals
  + Write signals
  + Signal notification ( Rx indication )
* Calibration interface
  + Read and write data to non-volatile storage.
* Diagnostic event interface
  + Notify diagnostic module of DTC statuses.

#### Timer access interface

The timer access interface can be used by functional blocks to access a free-running timer. The interface returns the number of milliseconds since power on. The 32-bit value will overflow which means that functional blocks must take this into account. The timer interface can be used in time sensitive operations where a high resolution is necessary. For functions that doesn’t require high resolutions the runnable can be used to determine the elapsed time.

The exact interface for timer access is specified in the BSP specification but it will follow the naming below:

GCL\_READ\_RTC\_Milliseconds(uint32 \*value);

# The Platform component layer

This layer consists of platform independent communication and support functions.

## GCL

The GCL layer is the key component of the architecture. It allows all the different parts of the system to communicate via standardized interfaces. The GCL layer is generated based on input from the other blocks in the architecture. The picture below shows the different parts of the GCL, and how they relate to the rest of the software architecture.

.

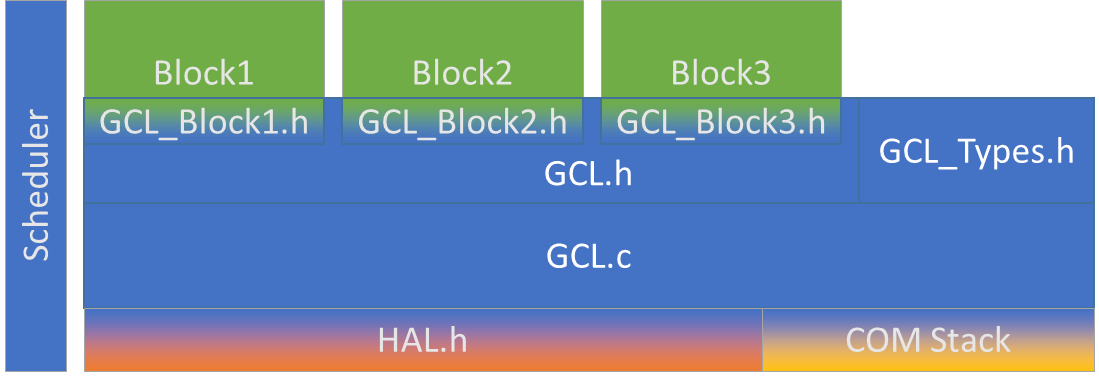


Figure 4 - The Generic Communication Layer

### Signal mapping

The FDT tool allows the cluster designer to define signal mappings between functions blocks and GCL signals. The GCL signals are defined either by other functions blocks or from standard components such as the BSP, CAN or Calibration interface. The designer creates signal mapping by connecting sinks from one block to a source of another blocks. Each source and sink is defined by a component name and a port name. The component name are either the name of a function block or “BSP”, while the port name is the signal name as defined in the Function Definition File or the BSP specification.

### The Signal Mapping File

The signal mapping XML file is used to connect function blocks to eachother as well as the BSP. The file consists of a number of mappings, each defined by a source and a sink component. Each source and sink is defined by a component name and a port name. The component name are either the name of a function block or “BSP”, while the port name is the signal name as defined in the Function Definition File or the BSP specification.

### Signal interface

The signal interface provides means to read, write, get receive notifications and signal timeout. Each signal interface is generated if a need for an access function exist (i.e a mapping exist).

#### Read interface

Each input signal defined will render a method call with the prototype:

uint8 GCL\_Read\_<NameOfFunctionBlock>\_<SignalName>(<SignalDataType> \*value);

#### Write interface

Each output signal defined will render a method call with the prototype:

void GCL\_Write\_<NameOfFunctionBlock>\_<SignalName>(<SignalDataType> value);

#### Signal update notification interface

For signals that have a signal reception notification the GCL will call the following interface in the function block.

void <NameOfFunctionBlock>\_<SignalName>\_Indication(void);

### Calibration interface

Each function blocks will define its run-time configuration parameters in its FDF file. Based on this information the Cal manager will generate a storage area for the parameter. The ID is defined in the FDT of the functionblock and the CAL manager will enumerated the ID. Therefor the functions blocks that use calibrations data shall include the file “CAL.h” to access the defined for the calibration ID.

The interfaces that shall be used to read and write that to the CAL manager is defined in chapter 3.5 Calibration manager

### Diagnostics interface

Function blocks will communicate with the diagnostics module directly via the Diag\_ReportError() interface. The interface is described in section 3.6 Diagnostic Manager

The types are define in the file DiagManager.h

## Scheduler

The scheduler is an integral part of the IC software as it handle the distribution of execution resources to all components. It calls the defined tasks from the configured function blocks and the pre-defined tasks for all platform software components.

### Scheduler and BSP software integration

The BSP startup code is responsible for initializing the scheduler during the initialization phase. After all the BSP initialization is complete the BSP shall call APP\_Run() which is the schedulers run function. Note that this function will never return, as the scheduler will manage the task execution.

### Task software interface

The interface between the scheduler and the function blocks consists of 3 functions that allow the functions to be initialized (init function), run (the runnable task) and for the function blocks to access an timer counter.

#### Init function

If specified in the FDF, the init-function will be called when the scheduler is initialized at power on. This call will occur before any cyclic function is called. The purpose of this function is to allow the function blocks to initialize to a known state. This can include loading calibration data, initializing variables etc.

The prototype for the initialization function is:

void *<function\_block\_name>*\_Init(void);

#### Run function (Task)

The runnable will be called from the scheduler according to the specification in the FDF.

The prototype for the runnable is:

void *<function\_block\_name>*\_Run(void);

### Scheduler configuration

The source file “Scheduler\_Configuration.c” is where the configuration for the scheduler can be found. The SchedulerTaskCfg\_t array defines all tasks to be executed by the scheduler. Each task is configured with an initial delay, a periodicity, the name of an initialization function, and the name of the function to periodically execute.

## Function Block Specific GCL header

Given the input and output signals defined in the Function Definition File, a function block specific header file is created. The name of the file will always be GCL\_<name of function block>.h. It is to be used by the function block to enable GCL communication.

### Input signals

Each input signal defined will render a method call with the prototype:

uint8 GCL\_Read\_<NameOfFunctionBlock>\_<SignalName>(<SignalDataType> \*value);

To acquire the data in the function block, define a variable of the signal’s data type and pass in a pointer to the function call. GCL will put the value in the pointer variable and return a status code to indicate whether the read was successful. E.g:

uint16 vehicleSpeed;

uint8 result = GCL\_Read\_VehicleSpeedGauge\_VehSpd(&vehicleSpeed);

### Output signals

Each output signal defined will render a method call with the prototype:

void GCL\_Write\_<NameOfFunctionBlock>\_<SignalName>(<SignalDataType> value);

To send data through GCL, call the function with the data to send. E.g:

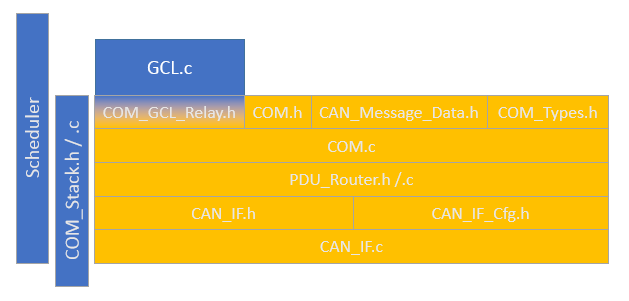
// Get the vehicle speed from an internal function

uint16 filteredVehicleSpeed = GetFilteredVehicleSpeed();

GCL\_Write\_VehicleSpeedGauge\_VehicleSpeedGaugeValue(filteredVehicleSpeed);

## COM stack

The COM stack in its entirety is part of both the platform component and the MCAL layers.



### COM

Interface from application towards the COM-stack. It routes signals between GCL and CAN. COM\_GCL\_Relay is a generated part that manages the data conversion between GCL and the COM stack.

### PDU Router

The PDU-Router routes messages between CAN, Application and a Diagnostic communication module (not yet implemented).

### CAN-IF

This is the interface to the CAN-driver. It abstracts different CAN-drivers and maintains the configuration of CAN-controller and mailboxes. Implemented in CAN\_IF.h/.c and configuration in CAN\_IF\_Cfg.h.

## Calibration manager

The calibration manager (CalMgr) is responsible for managing the runtime calibration of all blocks above the BSP. It will abstract the memory handling from the upper layers and use the memory interface in the BSP to store and retrieve data from non-volatile storage. The diagnostic manager will use the write interface in the calibration manager in order to store calibrations that is downloaded via test-tools. The function blocks will use the cal manager interface to read and write calibration data during runtime. The FB will call the Cal manager directly (i.e not via the GCL).

During startup the Cal manager will read all the parameters from the non-violate memory into RAM to minimize access time during runtime.

Based on the information in the included FDF files in the project the CAL manger will enumerate the calibration ID in the file CAL\_cfg.h

Note that all communication with the CAL manager shall be big endian.

### Cal retrieval interface

#### Generic Read interface

The interface below can be used to read data of arbitrary length from the CAL manager.

CAL\_STATUS CAL\_Read(uint16 u16ID, uint8\* pData, uint16 u16Length);

#### Type specific interfaces

In order to allow easy access the following interfaces are defined to read primitive datatypes:

CAL\_STATUS CAL\_Read8(uint16 u16ID, uint8\* u8Data);

CAL\_STATUS CAL\_Read16(uint16 u16ID, uint16\* u16Data);

CAL\_STATUS CAL\_Read32(uint16 u16ID, uint32\* u32Data);

### Cal write interface

The interfaces below will update the internal RAM variable as well as trigger a write to the NVRAM.

#### Generic write interface

The interface below can be used to write data of arbitrary length from the CAL manager.

CAL\_STATUS CAL\_Write(uint16 u16ID, uint8\* pData, uint16 u16Length);

#### Type specific interfaces

In order to allow easy access the following interfaces are defined to write primitive datatypes:

CAL\_STATUS CAL\_Write8(uint16 u16ID, uint8\* u8Data);

CAL\_STATUS CAL\_Write16(uint16 u16ID, uint16\* u16Data);

CAL\_STATUS CAL\_Write32(uint16 u16ID, uint32\* u32Data);

## Diagnostic Manager

*<Note: Since this section is DETC responsibility I will only describe the interfaces and the expected behavior related to the specified interfaces.>*

The Diagnostic module (DM) is responsible for managing the diagnostic events, reading and writing calibration via the CAN interface and the diagnostic communication (via the COM stack).

The diagnostic modules have four types of interfaces:

1. Task interface
2. Diagnostic event interface
3. Calibration interface
4. Communication interface

The interfaces and their expected behavior are described in the following subchapters.

### Task interface

The task interface consists of the initialization function and the scheduled runnable(s). The interface function prototypes are:

void Diag\_Init(void)

void Diag\_Run(void)

### Diagnostic event interface

The diagnostic event interface exposes a interface to allow function blocks and other platform component to set DTC status in the diagnostic module. The calling function (Function blocks) will report a status change of the DTC i.e. if the status of a DTC has change from inactive to active the FB will call the diagnostic module with the DTC number and status set to active. In the same way the FB report to the DM if an error has been healed (i.e. the condition to set the DTC as active is no longer present). The DM is responsible for handling the status communication with the off-board tester i.e the DM is responsible for managing the setting of the DTCStatusMask (ref. to ISO 14229).

#### Software interface

The function blocks shall use the following interface to report an error to the DM.

DIAG\_ReportError(tDTCRecord);

##### tDiagReport definition

The type used in the ReportError interfaces is defined as below:

typedef enum

{

DTC\_ACTIVE,

DTC\_INACTIVE

} tDTCStatus;

Typedef struct

{

uint32 u32DTCNumber;

tDTCStatus Status;

} tDiagReport;

The types for this interface shall be define in “DiagManager.h”

### Calibration interface

The diagnostic module manages the transmission of calibration data to/from the ECU via UDS server $22 and $2E. The DM can access the calibration values stored in the calibration manager via the CALMGR interfaces. The CALMGR will then manage the read/write of the data to NVRAM.

The interfaces that shall be used to read and write that to the CAL manager is defined in chapter 3.5 Calibration manager

### Communication interface

The DM will utilize the PDUR interfaces in order to be able to receive and transmit CAN messages. This chapter will describe the interface between the DM and the PDUR.

#### RX messages

Messages received form the CAN bus will be forwarded to the following interface of the Diagnostic module:

void Diag\_RxIndication(uint32 u32CanIfPduId,uint8\* u8DataPtr,uint8 u8DataLen);

#### TX Messages

The DM shall use the following interface to transmit data.

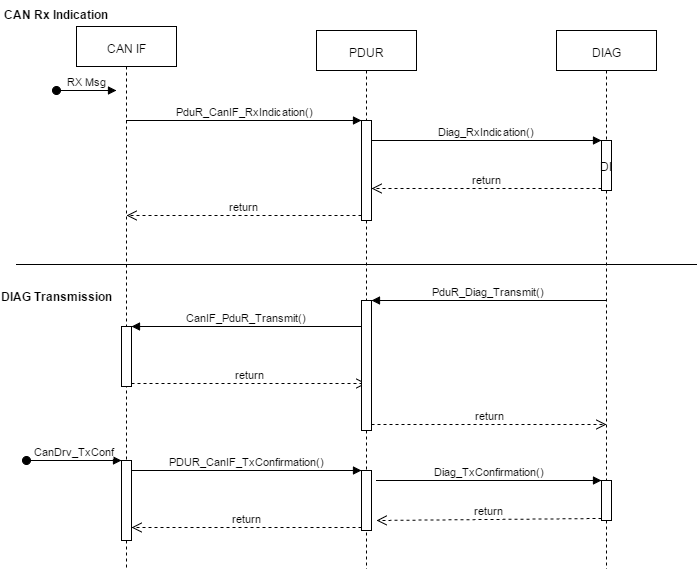
void PduR\_Diag\_Transmit(uint32 u32CanIfPduId,uint8\* u8DataPtr,uint8 u8DataLen);

#### TX Confirmation

In order to manage correct transmission of TP data the diagnostic module require a transmission confirmation when the CAN driver module has successfully transmitted a message. The indication will be propagated thru the COM stack to the following interface:

void Diag\_TxConfirmation(uint32 u32CanIfPduId);

#### Sequence diagram for RX and Tx

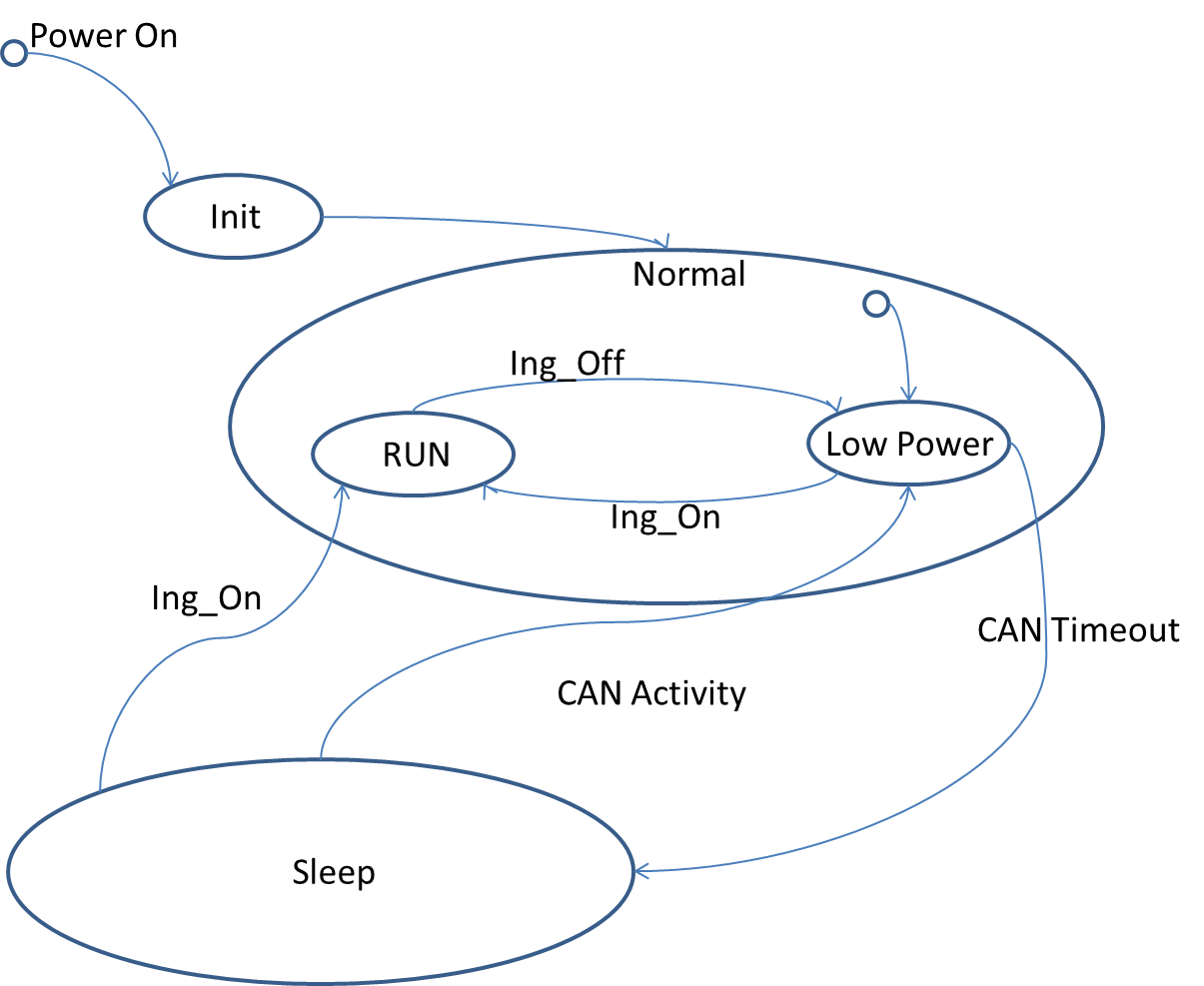


## Mode Manager

The mode manager (MM) is responsible for managing the mode of the ECU. It interacts with the BSP and the platform components to determine and set the correct mode. The mode is propagated to the application layer via the GCL. The MM communicates with the BSP via requests to change mode and BSP reports back to the MM via the MM\_NotifyModeChange(newMode) interface.

### State chart

The chart below describes the states and transitions that are used within the mode manager. There are 3 main states: INIT, NORMAL and SLEEP. The INIT mode is only reached when the IC is powered on. The NORMAL mode is the mode where the software is running. SLEEP mode is the halt or stop mode where no software are run.



### Propagation of ECU Mode:

After the mode manager have successfully transitioned to a new state mode it will set the system to a correspondent power mode via the GCL interface GCL\_Write\_Platform\_PowerMode(powerMode).  
Other parts that need to know the system power mode will then get what power mode the system is in via their PowerMode signal sink.

The power modes are defined as:

#define POWER\_LOW 0x00

#define POWER\_IGNITION\_OFF 0x01

#define POWER\_IGNITION\_ON 0x02

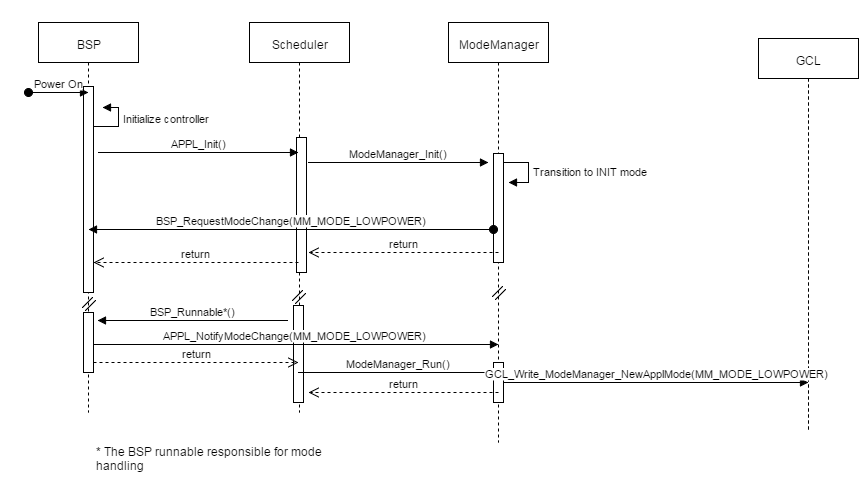
Application blocks and platform components that are required to operate in modes other that IGN\_ON must listen to these signals and take actions accordingly.

### General transition behavior

#### Power up sequence

The image below shows the power on sequence for the BSP and mode manager.

In general the BSP is started and initializes the controller. It also initializes the scheduler, which in turn will initialize the mode manager which will be initialized to INIT mode. NOTE: The Mode Manager initialize the CAN transceiver and starts the CAN communication.

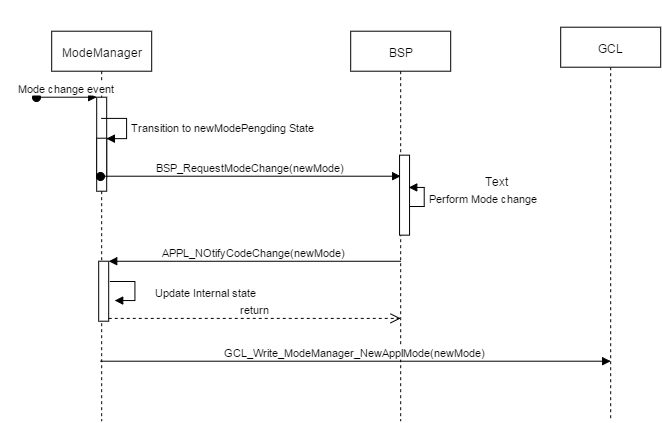


#### Mode change initiated from Mode manager

The mode manage will perform the following actions when changing state.

1. Request a ECU mode change via the BSP\_RequestModeChange(newMode) interface.
2. Transition to a MM-internal wait state. It will remain in the internal wait state until conformation of the new state is received from BSP via Appl\_ModeChangeNotification(newMode)

The sequence diagram below shows the generic sequence when the mode manager initialize a mode change



### Mode descriptions

The MM has 2 main states: “NORMAL” and “SLEEP” mode. In addition to these two states an transition state: “INIT” is defined. The INIT state is reached after power on. The following chapters contain a detailed explanation of the states, triggers and events for the main states and the sub states of the main states.

#### Init

The Init statet is defined to allow the ECU to start and configure the perphials of the MCU. The states is entered upon power on and left when all the necessary initialization is done. The mode manager will reach this state when a call to ModeManager\_Init() is called. Otherwise this state is internal to the BSP and the startup code of the ECU.

#### Normal:

The NORMAL mode contains two sub states: **Low\_power** and **Run.** The **low\_power** mode is the default entry point in the NORMAL state. The low power mode is used to reduce the power requirements while maintaining some functionality. The RUN mode is the full operational mode where the IC normally is during runtime.

The Mode manager transitions to normal mode, sub state **low\_power** after power on.

##### Substate: low\_power.

The **Low\_power** mode is the default transition when the mode manager reaches normal mode. In this mode most of the peripherals of the MCU is disabled. Upon entering the **low\_power** state the mode manager will send a request to the BSP to transition to the **low\_power** mode. The application function blocks and platform components that shall be run in ING\_OFF state shall monitor the GCL mode management signal to determine if the low power functionality shall be active. The **low\_power** mode can be left in two ways: To SLEEP mode via a CAN timeout or to RUN mode via IGN\_ON.

Transition to RUN mode:

The State will change from low power mode to RUN mode if ING\_ON is received.

Transition to SLEEP:

The modemanger will change to SLEEP mode if no CAN communication has been received in <tbd> time. The CAN stack will monitor the traffic and notify the mode manager is no traffic has been received within the defined time. The CAN stack will set the CAN transceiver to sleep mode before notifying the mode manager which in turn will initiate the transition to sleep mode.

##### Substate: RUN

The RUN mode is the normal operating mode where all the software is run. The mode is reached if the ignition is turned on. The RUN mode can be reached either from SLEEP or from **low\_power** mode.

Transition to low power.

#### SLEEP

The sleep mode can only be reached when the ECU is in low power mode and there is no activity on the CAN bus. The CAN stack will notify the MM that a CAN timeout has occurred and the MM will request the mode change from the BSP. The SLEEP mode is halt or stop mode of the MCU.

##### Transition to low\_power:

Transition from SLEEP will be performed by the BSP when an external interrupt from the CAN transceiver is received. If the CAN transceiver detects traffic on the CAN bus is will notify the MCU via a discrete input. The BSP shall inform the MM via the APPL\_WakeupNotify(CAN) interface that the reason for wakeup is the CAN communication. The BSP shall then transition to **low\_power** and notify the mode manager via the APPL\_NotifyModeChange() interface. The MM will notify the CAN stack that will request a mode change of the CAN transceiver.

##### Transition to run:

The transition from SLEEP to **run** is triggered via an external interrupt that is connected to the ING\_ON wire. If this interrupt is received durinig sleep mode the BSP shall transition to normal run mode(MCU mode) notify the MM via the APPL\_WakeupNotify() and APPL\_NotifyModeChange() when the transition is complete.

### Software Interfaces:

#### Task interface

ModeManager\_Init()

Modemanager\_Run()

#### BSP Interface description

##### Interfaces exposed to the BSP

This section contains the interfaces that the BSP is expected to call.

Appl\_WakeupNotify(source)

Appl\_NotifyModeChange(BSP\_ECUModes)

##### Required BSP interfaces

These interfaces are required to be implemented in the BSP in order for the MM to operated correctly.

BSP\_SetTranceiverMode()

BSP\_RequestModeChange()

#### CAN Stack interface

##### Interface exposed to the CAN stack

This interface shall be used by the CAN stack to notify the MM that no CAN communication has been received within a specified time.

ModeManager\_CAN\_Timeout();

# The Platform Layer

The platform layer consists primarily of the Board Support Package and CAN drivers. This is hardware specific and will have to be adapted depending on platform. There will also be a PC-BSP used for simulation and testing purposes.

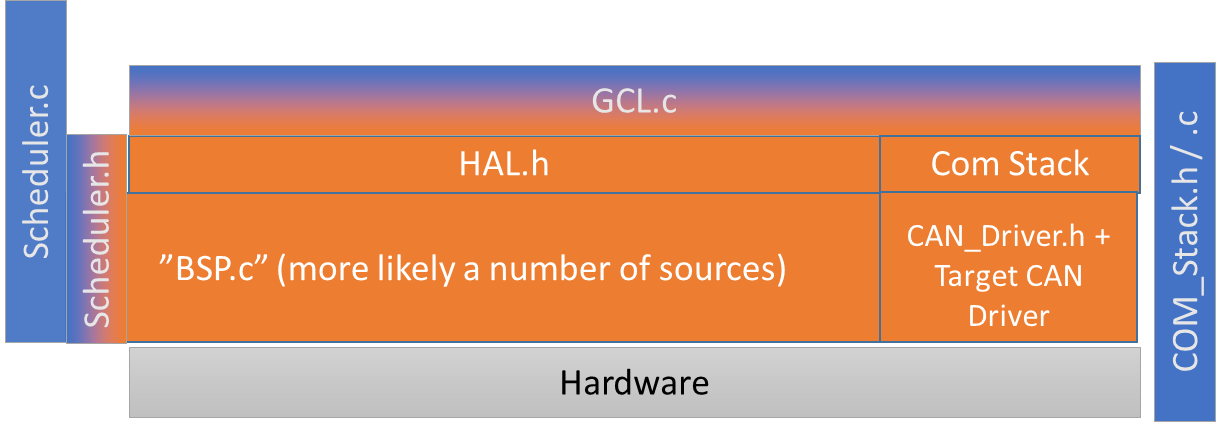


Figure 5 - The Platform Layer

## HAL layer

The Hardware abstraction layer is used to provide the GCL with a signal interface to interact with the BSP. This is achieved by abstracting the indexes of the individual inputs/outputs to a defined signals name (e.g. digital input with index 0x02 is abstracted into signal HighBeamSwitch\_Active() )

## BSP Interface

The BSP interface (BSP.h) is generated from the BSP specification XML. It is merely a listing of all functions that should be implemented in the BSP, either for target or for PC simulation. The BSP will also initiate and run the scheduler using functions in Scheduler.h

## Target BSP

The Board Support Package implementation is hardware specific and not a part of this document. It is important that the implementation fulfills the BSP.h interface and initializes the scheduler.

## PC-BSP (Simulator)

The PC-BSP will provide a mock BSP with in-memory representation of signal values and hardware states. The mock also provides a network socket to enable manipulation and inspection.

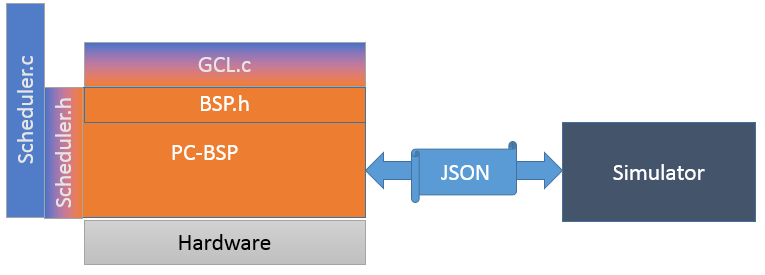


Figure 6 - PC-BSP

## CAN Driver (Target/Simulator)

There are different CAN-drivers for target and simulation. The simulated CAN-driver (vector\_can\_driver.c) is based on Vector’s XL-driver, a virtual CAN-interface.

### CAN-Driver

The CAN driver manages the CAN controller of the MCU. It is responsible for managing the CAN controller hardware during both startup and runtime. The startup configures the timing of the CAN bus, message boxes and filtering etc. During runtime the CAN driver manages transmit and receive queues as well as mode management and error management.

# Bootloader

The bootloader is used to allow updates of the software after production of the device and the OEM vehicle. The bootloader manages the download and flash programming of the MCU. It implements the UDS services that are needed to communicate with the off board tester e.g $3E, $34, $36 etc.

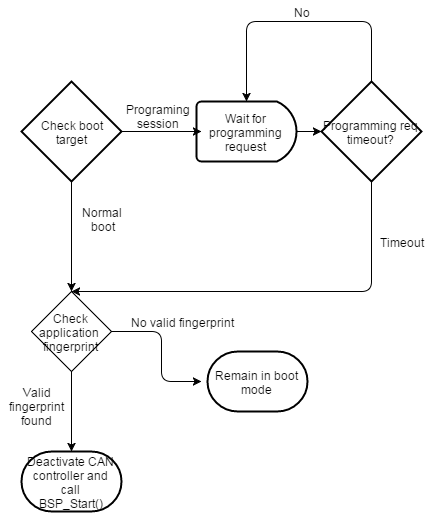
## MCU startup sequence

*<note: As the bootloader is delivered by T-engineering this chapter and information stated here shall be reviewed by both T-engineering and DETC>*

The bootloaders main() function is the entry point of the MCU after power on. The bootloader will therefore initialize the peripherals that are necessary to download a new SW.

The bootloader will then check a special registry <tbd> to determine if the MCU was reset as part of a programming session ( Initiate diagnostic session, Programming mode: UDS service $10 Subparameter $02). If the registry is set to a specific value <tbd> the bootloader shall remain in bootloader mode until a programming request has been received of a timeout has occurred.

If a programming session was not requested, it will then check if an valid software are present in the flash memory. This in performed by checking a specified address for a fingerprint. This fingerprint shall also be stored inverted in a separate memory address in order to increase robustness.



## Interface Bootloader and BSP

The interface between the bootloader and BSP consist of a 2 defined memory addresses, one registry and one function.

### Memory addresses

These variables are store in flash memory and shall be written as the absolute last part of the application software during a programming event. This is to ensure that the entire application has been downloaded before a valid fingerprint is written to the memory.

|  |  |  |
| --- | --- | --- |
| Variable name | Variable datatype | Address |
| u32ApplFingerprint | uint32 | <tbd> |
| U32InvertedApplFingerprint | uint32 | <tbd> |

The fingerprint for a valid application shall be 0xABBA.

### Boot target registry

If a programming session has been requested via a diagnostic requests (Initiate diagnostic session, programming session) the diagnostic manager shall write the boottarget ID to a special registry of the MCU. This registry and value are defined below.

|  |  |  |
| --- | --- | --- |
| Registry | Normal boot mode value | Programming session value |
| <tbd> | <tbd> | <tbd> |

### BSP start function

If the bootloader has determined that normal application shall be started the bootloader will de-initialize the CAN controller and call the BSP function: void BSP\_Start(void);

# Software Structure

## Generated software structure

Generating the software results in the destination folder becoming populated with a number of files and folders. The root output folder will contain the GNU Makefile and three major subfolders.

|  |  |
| --- | --- |
| The Application Folder The purpose of the application folder is to contain all function block specific code. The platform The platform folder contains target independent code that tie together the function blocks with the BSP, as well as support functions. The MCAL folder A part from the standard subfolder called “common”, where the interfaces to the MCAL are gathered, there is also a subfolder for each target.  Although there is only a PCSim subfolder currently, as more hardware targets are supported, they will have their hardware specific software adaptations in subfolders here. | TODO:  Insert mage of new folder structure  Figure 7 - Generated Folder Structure |

### The Makefile

The makefile consists of instructions for the GNU Make tool on how to build the IC software. The file is generated automatically and editing the file manually should never be required.

## Linker considerations

As the bootloader will be the first entry point after power on the bootloader should be located in the beginning of the memory. Therefore, the BSP and application software should be allocated to another address. The starting address of the application software shall be: <tbd>

# Glossary

|  |  |
| --- | --- |
| BSP | Board Support Package, the intermediate software layer between the generic platform code and the hardware. |
| GCL | Generic Communication Layer, the intra-component communication layer in the platform code. |
| FDF | Function Definition File, an XML file in which metadata about a function block is described. |
| GNU Make | A tool, which controls the generation of executables and other non-source files of a program from the program’s source files. (http://www.gnu.org/software/make/) |
| IC software | Instrument Cluster software |
| XML | eXtensible Markup Language, a W3C standard for describing information in a human- and machine-readable format |