|  |  |
| --- | --- |
|  | |
| Application Programming Interface | |
| **Version:** | 1.0 |
| **Date:** | 2020-01-11 |

# GENERAL REMARKS

The target of this Product Specification is to support Product Engineering for Software development.

This document is referring to a ARM Based device, implementing a dedicated SW communicating over a LIN bus.

[FAULTS TABLE] contains the whole list of faults signaled to the Master ECU for this sample.

[PARAMETERS TABLE] contains the whole relevant software parameters established for this sample.

# REVISION HISTORY

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Rel.*** | ***Date*** | ***Description*** | ***Sections affected*** | ***Prepared by*** | ***Checked by*** | ***Approved by*** |
| 1.0 | 1/20/2020 | First Draft | All | C. Hiler | TBD | TBD |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

# REFERENCE

|  |  |  |  |
| --- | --- | --- | --- |
| **Ref** | **Revision** | **Document** | **Source** |
| [Ref 1] |  |  |  |
| [Ref 2] |  |  |  |
| [Ref 3] |  |  |  |
| [Ref 4] |  |  |  |
| [Ref 5] |  |  |  |
| [Ref 6] |  |  |  |
| [Ref 7] |  |  |  |
| [Ref 8] |  |  |  |
| [Ref 9] |  |  |  |
| [Ref 10] |  |  |  |
| [Ref 11] |  |  |  |
| [Ref 12] |  |  |  |
| [Ref 13] |  |  |  |
| [Ref 14] |  |  |  |
| [Ref 15] |  |  |  |
| [Ref 16] |  |  |  |
| [Ref 17] |  |  |  |

TABLE OF CONTENTS

[GENERAL REMARKS 2](#_Toc31706064)

[REVISION HISTORY 3](#_Toc31706065)

[REFERENCE 4](#_Toc31706066)

[LIST OF TABLES 10](#_Toc31706067)

[No table of figures entries found.LIST OF FIGURES 10](#_Toc31706068)

[1. PURPOSE 12](#_Toc31706069)

[1.1. SCOPE OF DOCUMENT 12](#_Toc31706070)

[1.2. DOCUMENT EXTRACT 12](#_Toc31706071)

[1.3. CONFIDENTIALITY 12](#_Toc31706072)

[2. INTRODUCTION 13](#_Toc31706073)

[2.1. MOTOR DRIVE (System Architecture) 14](#_Toc31706074)

[2.2. FUNCTIONAL DESCRIPTION 14](#_Toc31706075)

[2.3. Slide-Out Motor Software Description 14](#_Toc31706076)

[2.4. SCOPE OF DELIVERY 15](#_Toc31706077)

[2.5. EXAMPLE INTERFACE FUNCTION TABLE 16](#_Toc31706078)

[3. REFERENCE DOCUMENTS 17](#_Toc31706079)

[4. MEMORY MANAGEMENT 17](#_Toc31706080)

[4.1. BASE SOFTWARE MEMORY 17](#_Toc31706081)

[4.1.1. VOLATILE RAM 18](#_Toc31706082)

[4.1.2. NON-VOLATILE RAM 18](#_Toc31706083)

[4.1.3. GUARDED NON-VOLATILE RAM 18](#_Toc31706084)

[4.1.4. DMA RAM 18](#_Toc31706085)

[4.1.5. CONST DATA 19](#_Toc31706086)

[4.1.6. EEPROM DATA 19](#_Toc31706087)

[4.1.7. CODE ROM 19](#_Toc31706088)

[4.1.8. RAM/ROM USAGE REPORTING 19](#_Toc31706089)

[4.2. NVRAM MANAGEMENT 20](#_Toc31706090)

[4.3. EEPROM MANAGEMENT 20](#_Toc31706091)

[4.3.1. INTERFACE STRUCTURE 20](#_Toc31706092)

[4.3.2. HW\_InitializeEEMimicRAM () 21](#_Toc31706093)

[4.3.3. HW\_InitiateEEPROMWrite () 22](#_Toc31706094)

[4.3.4. HW\_EEWritesAreComplete () 23](#_Toc31706095)

[4.4. DELIVERABLES 24](#_Toc31706096)

[4.4.1. OBJECT FILES 24](#_Toc31706097)

[4.4.2. HEADER FILES 24](#_Toc31706098)

[4.4.3. SOURCE CODE 25](#_Toc31706099)

[4.4.4. DOCUMENTATION 25](#_Toc31706100)

[5. INTERRUPTS & EXCEPTIONS 25](#_Toc31706101)

[5.1.1. MINIMUM INTERRUPT REQUIREMENTS 25](#_Toc31706102)

[5.2. IRQ PIN INTERRUPTS – EDGE DETECTS 26](#_Toc31706103)

[5.2.1. INTERFACE STRUCTURE 26](#_Toc31706104)

[5.2.2. HW\_EdgeDetectInit () 26](#_Toc31706105)

[5.2.3. HW\_EdgeWasDetected () 27](#_Toc31706106)

[5.2.4. HW\_EdgeDetectPinIsLow () 27](#_Toc31706107)

[5.2.5. HW\_EdgeDetectPinIsHigh () 28](#_Toc31706108)

[5.2.6. HW\_EdgeDetectInterruptEnable () 28](#_Toc31706109)

[5.2.7. HW\_EdgeDetectInterruptDisable () 29](#_Toc31706110)

[5.3. POWER SUPPLY LOW VOLTAGE DETECTION 30](#_Toc31706111)

[5.3.1. HW\_PSUndervoltageFallingEdgeHandler() 30](#_Toc31706112)

[5.3.2. HW\_PSUndervoltageRisingEdgeHandler() 31](#_Toc31706113)

[5.4. ENABLING AND DISABLING INTERRUPTS 31](#_Toc31706114)

[5.5. WATCHDOG SERVICING 32](#_Toc31706115)

[5.6. DELIVERABLES 33](#_Toc31706116)

[5.6.1. OBJECT FILES 33](#_Toc31706117)

[5.6.2. HEADER FILES 34](#_Toc31706118)

[5.6.3. SOURCE CODE 34](#_Toc31706119)

[5.6.4. DOCUMENTATION 34](#_Toc31706120)

[6. CONTROLLED RESET 34](#_Toc31706121)

[6.1. HW\_SystemReset () 35](#_Toc31706122)

[6.2. HW\_SoftReset () 35](#_Toc31706123)

[6.3. DELIVERABLES 36](#_Toc31706124)

[6.3.1. OBJECT FILES 36](#_Toc31706125)

[6.3.2. HEADER FILES 36](#_Toc31706126)

[6.3.3. SOURCE CODE 37](#_Toc31706127)

[6.3.4. DOCUMENTATION 37](#_Toc31706128)

[7. ANALOG TO DIGITAL CONVERSIONS 37](#_Toc31706129)

[7.1. SW TRIGGERED A2D 37](#_Toc31706130)

[7.2. HW\_InitA2D () 38](#_Toc31706131)

[7.3. HW\_GetRawA2D () 39](#_Toc31706132)

[7.4. HW\_A2D\_Trigger () 39](#_Toc31706133)

[7.5. HW\_A2D\_Completed () 40](#_Toc31706134)

[7.6. HW\_A2D\_GetConfigScale () 41](#_Toc31706135)

[7.7. HW\_A2D\_RequestCalibration () 41](#_Toc31706136)

[7.8. DELIVERABLES 42](#_Toc31706137)

[7.8.1. OBJECT FILES 42](#_Toc31706138)

[7.8.2. HEADER FILES 43](#_Toc31706139)

[7.8.3. SOURCE CODE 43](#_Toc31706140)

[7.8.4. DOCUMENTATION 43](#_Toc31706141)

[8. DIGITAL INPUTS 43](#_Toc31706142)

[8.1. INTERFACE STRUCTURE 43](#_Toc31706143)

[8.2. HW\_UpdateDigInPinState () 43](#_Toc31706144)

[8.3. HW\_GetDigInPin\_InstState () 44](#_Toc31706145)

[8.4. DELIVERABLES 45](#_Toc31706146)

[8.4.1. OBJECT FILES 45](#_Toc31706147)

[8.4.2. HEADER FILES 45](#_Toc31706148)

[8.4.3. SOURCE CODE 45](#_Toc31706149)

[8.4.4. DOCUMENTATION 46](#_Toc31706150)

[9. DIGITAL OUTPUTS 46](#_Toc31706151)

[9.1. INTERFACE STRUCTURE 46](#_Toc31706152)

[9.2. HW\_TurnOnDigOut () 47](#_Toc31706153)

[9.3. HW\_TurnOffDigOut () 48](#_Toc31706154)

[9.4. HW\_DigOutIsFailing() 48](#_Toc31706155)

[9.5. DELIVERABLES 49](#_Toc31706156)

[9.5.1. OBJECT FILES 49](#_Toc31706157)

[9.5.2. HEADER FILES 50](#_Toc31706158)

[9.5.3. SOURCE CODE 50](#_Toc31706159)

[9.5.4. DOCUMENTATION 50](#_Toc31706160)

[10. PWM OUTPUTS 51](#_Toc31706161)

[10.1.1. Autonertia PWM Interface Structure 51](#_Toc31706162)

[10.1.2. 3 Phase Inverter Driver with Dead Time Insertion 52](#_Toc31706163)

[10.2. HW\_InitPWM () 54](#_Toc31706164)

[10.3. HW\_SetPWMFreq () 55](#_Toc31706165)

[10.4. HW\_SetPWM\_DC ( ) 56](#_Toc31706166)

[10.5. HW\_TurnOnPWM() 57](#_Toc31706167)

[10.6. HW\_TurnOffPWM() 57](#_Toc31706168)

[10.7. HW\_PWMIsFailing ( ) 58](#_Toc31706169)

[10.8. HW\_SetPWMPolarity ( ) 59](#_Toc31706170)

[10.9. DELIVERABLES 59](#_Toc31706171)

[10.9.1. OBJECT FILES 59](#_Toc31706172)

[10.9.2. HEADER FILES 60](#_Toc31706173)

[10.9.3. SOURCE CODE 60](#_Toc31706174)

[10.9.4. DOCUMENTATION 60](#_Toc31706175)

[11. PERIOD AND PULSE WIDTH MEASUREMENTS 61](#_Toc31706176)

[11.1. INTERFACE STRUCTURE 61](#_Toc31706177)

[11.2. HW\_InitPeriodMeasurement () 62](#_Toc31706178)

[11.3. HW\_InitPWMeasurement () 62](#_Toc31706179)

[11.4. HW\_InitDCMeasurement () 63](#_Toc31706180)

[11.5. HW\_GetPWMPeriod () 64](#_Toc31706181)

[11.6. HW\_GetPulseWidth () 64](#_Toc31706182)

[11.7. HW\_GetPWMDutyCycle () 65](#_Toc31706183)

[11.8. HW\_PeriodHasCompleted () 66](#_Toc31706184)

[11.9. HW\_PulseHasCompleted () 67](#_Toc31706185)

[11.10. HW\_DCHasCompleted () 67](#_Toc31706186)

[11.11. HW\_PWMInputPinIsHigh () 68](#_Toc31706187)

[11.12. HW\_PWMInputPinIsLow () 69](#_Toc31706188)

[11.13. DELIVERABLES 69](#_Toc31706189)

[11.13.1. OBJECT FILES 69](#_Toc31706190)

[11.13.2. HEADER FILES 70](#_Toc31706191)

[11.13.3. SOURCE CODE 70](#_Toc31706192)

[11.13.4. DOCUMENTATION 70](#_Toc31706193)

[12. DIRECT MEMORY ACCESS (DMA) 70](#_Toc31706194)

[12.1. DMA GLOBAL REGISTERS 71](#_Toc31706195)

[12.1.1. DMA Control Register (Example) 71](#_Toc31706196)

[12.2. DMA Enable Error Interrupt Register (EDMA\_EEIRH, EDMA\_EEIRL) 71](#_Toc31706197)

[12.3. DMA Enable Request Register (EDMA\_ERQRH, EDMA\_ERQRL) 71](#_Toc31706198)

[12.4. DMA Interrupt Request Register (EDMA\_IRQRH, EDMA\_IRQRL) 71](#_Toc31706199)

[12.5. DMA Error Register (EDMA\_ERH, EDMA\_ERL) 71](#_Toc31706200)

[12.6. DELIVERABLES 72](#_Toc31706201)

[12.6.1. OBJECT FILES 72](#_Toc31706202)

[12.6.2. HEADER FILES 72](#_Toc31706203)

[12.6.3. SOURCE CODE 72](#_Toc31706204)

[12.6.4. DOCUMENTATION 73](#_Toc31706205)

[13. CONTROLLER SPECIFIC HW DIAGNOSTICS 73](#_Toc31706206)

[13.1. HW\_ControllerDiagnostic1 () 73](#_Toc31706207)

[13.2. HW\_ControllerDiagnostic2 () 74](#_Toc31706208)

[13.3. HW\_ControllerDiagnostic3() 75](#_Toc31706209)

[13.4. HW\_ControllerDiagnostic4() 75](#_Toc31706210)

[13.5. Controller Cause of Reset 76](#_Toc31706211)

[13.6. HW\_InitCauseOfReset() 77](#_Toc31706212)

[13.7. HW\_CauseOfReset() 78](#_Toc31706213)

[13.8. DELIVERABLES 78](#_Toc31706214)

[13.8.1. OBJECT FILES 78](#_Toc31706215)

[13.8.2. HEADER FILES 79](#_Toc31706216)

[13.8.3. SOURCE CODE 79](#_Toc31706217)

[13.8.4. DOCUMENTATION 79](#_Toc31706218)

[14. TASK MANGEMENT – INITIALIZATION, RUNNING, AND POWERDOWN 79](#_Toc31706219)

[14.1. Clock Frequency 79](#_Toc31706220)

[14.2. HW\_GetSysClockFrequency () 80](#_Toc31706221)

[14.3. Module Initialization 80](#_Toc31706222)

[14.4. Temporal Tasks 81](#_Toc31706223)

[14.5. Powerdown 82](#_Toc31706224)

[14.6. Miscellaneous Functions 82](#_Toc31706225)

[14.7. HW\_PowerUpInit () 82](#_Toc31706226)

[14.8. HW\_WasBatteryDisconnected () 83](#_Toc31706227)

[14.9. HW\_PSInitializationStatus () 84](#_Toc31706228)

[14.10. HW\_PSStatus () 84](#_Toc31706229)

[14.11. HW\_TurnPowerSupplyOff () 85](#_Toc31706230)

[14.12. HW\_APIVersion () 86](#_Toc31706231)

[14.13. DELIVERABLES 86](#_Toc31706232)

[14.13.1. OBJECT FILES 86](#_Toc31706233)

[14.13.2. HEADER FILES 87](#_Toc31706234)

[14.13.3. SOURCE CODE 87](#_Toc31706235)

[14.13.4. DOCUMENTATION 87](#_Toc31706236)

[15. BOOTLOADER INTERFACES 87](#_Toc31706237)

[15.1. Interface Structure 88](#_Toc31706238)

[15.2. BL\_InitializeEEMimicRAM () 88](#_Toc31706239)

[15.3. BL\_InitiateEEPROMWrite () 89](#_Toc31706240)

[15.4. BL\_PowerUpInit () 90](#_Toc31706241)

[15.5. BL\_TurnPowerSupplyOff () 91](#_Toc31706242)

[15.6. BL\_WasBatteryDisconnected () 91](#_Toc31706243)

[15.7. BL\_SystemReset () 92](#_Toc31706244)

[15.8. BL\_APIVersion () 93](#_Toc31706245)

[15.9. DELIVERABLES 93](#_Toc31706246)

[15.9.1. OBJECT FILES 93](#_Toc31706247)

[15.9.2. HEADER FILES 94](#_Toc31706248)

[15.9.3. SOURCE CODE 94](#_Toc31706249)

[15.9.4. DOCUMENTATION 94](#_Toc31706250)

[16. PROJECT SPECIFIC INFORMATION 95](#_Toc31706251)

[16.1. Directory Structures 95](#_Toc31706252)

[16.2. Header File Naming Convention 95](#_Toc31706253)

[16.3. Libraries 96](#_Toc31706254)

[16.4. General Comments on Deliverables 96](#_Toc31706255)

# LIST OF TABLES

# No table of figures entries found.LIST OF FIGURES

Figure 1 – MeWP generic design: acceptable Valve/Actuator orientation 12

Figure 2 – ARM Based Device Valve/Actuator pin-out 15

Figure 3 – ARM Based Device Inrush current profile at ECU power on transition (2.5 mm2 wire harness section, harness length 2.2m) 16

Figure 4 – ARM Based Device Inrush current profile at ECU power on transition (2.5 mm2 wire harness section, harness length 3m) 17

Figure 5 – ARM Based Device Inrush current profile at ECU power on transition (2.5 mm2 wire harness section, harness length 4m) 18

Figure 6 – ARM Based Device Valve/Actuator Mating Part 20

Figure 7 – ARM Based Device Valve/Actuator dimensions 21

Figure 8 – ARM Based Device Package 21

Figure 9 – MeWP block diagram 23

Figure 10 – ARM Based Device LIN Typical characteristic curves. 24

Figure 11 – Motor power stage 25

Figure 12 - LIN cluster diagram example 26

Figure 13 – MeWP (EWP) relevant Frames in Schedule Table 26

Figure 14- ARM Based Device (EWP) Node Slave Configuration 27

Figure 15 – Diagnostic session (ECU Reprogramming) 27

Figure 16 – ECU behaviour and timings 37

Figure 17 – MeWP Relationship: Master ECU request vs. Motor speed 40

Figure 18 – Temperature Protection strategy activation thresholds 44

Figure 19 - Temperature Protection strategy healing thresholds 44

# PURPOSE

## SCOPE OF DOCUMENT

This document reflects the application programming interface to supplier provided software

components that will be used in the development of Autonertia management software for an ARM Based Device.

|  |  |  |  |
| --- | --- | --- | --- |
| Customer P/N Assembly | Customer P/N  Valve/Actuator only | Autonertia P/N | Description |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## DOCUMENT EXTRACT

This document is an API spec relating to the programming of ARM Based Devices for Autonertia, Inc.

## CONFIDENTIALITY

This product characterization document is intended to be a working document for departments involved in the project. Transfer to third parties is governed by stipulations in the development contract concerning confidentiality.

Disclosure to third parties is prohibited without agreement of Autonertia, Inc.

# INTRODUCTION

The Autonertia Application Programming Interface (API) has been developed to provide a standard method for Autonertia application software to interface to various controller hardware. The API has been designed to impose a minimal number of constraints on the developers implementing the API, and yet provide Autonertia with a means of maintaining portability of its control algorithms across hardware platforms.

The following diagram generalizes the expected architecture. It is by no means comprehensive of every interface/requirement that needs to be developed, but provides an indication as to the intent.



## MOTOR DRIVE (System Architecture)

The Motor Drive consists of:

* Mechanical parts (Gears, Stator, Housing, Cover)
* Motor (Rotor, Stator)
* Control board (electronic parts)
* Software

## FUNCTIONAL DESCRIPTION

The prodcut is a Slide-Out controller driven by a brushless electric motor.

Whenever the motor drive receives a correct command, the electronic control unit processes it and drives the motor at desired speed until the final position is achieved.

## Slide-Out Motor Software Description

There are three layers to the Autonertia software architecture. The HWIO layer is responsible for interfacing directly to the hardware. The Basic Software Layer is the abstraction of all hardware signals. The Application layer consists of the implementation of all algorithms and strategies of the Slide-Out Control application.

The HWIO layer is responsible for providing A2D conversion results, driving output pins, reading digital pins, sending and receiving LIN information, managing interrupts responses, reading PWM inputs and driving PWM outputs. The HWIO layer does not interface directly with application software or the diagnostic management system. The HWIO layer software

components will be based on I/O pins, raw signal data, etc. It will have to provide interfaces to any initialization or temporal processing functions necessary to support the devices for the operating system to invoke them.

The second layer of the Autonertia software architecture is the Basic Software Layer.

Examples of objects in this layer include:

* Ignition R/S switch – debouncing signal, providing ON/OFF and switch state timer information.
* Temperature – converting A2D channel to a scaled representation, diagnostics.
* Solenoid – setting up the frequency and driving out a set duty cycle over PWM output, diagnostics.
* RPM – converting period time measurements from the API layer into RPM representations at various scalings.

At the top of the architecture is the Application layer. It consists of all motor control and or other controller related application software, including power moding, and diagnostics.

Each chapter of this document deals with a specific aspect of the software that is necessary to interface the HWIO layer to the rest of the Autonertia software. The topic is discussed in general terms explaining the overall requirement. Then, the exact interface functions are defined. Each interface function is defined in a chart as depicted below. Note that all fields of the chart are not applicable to all interface functions.

In many cases, the interface is realized using function arguments that will be used to identify instances of the physical devices. For example, simple digital outputs will be controlled using a pointer to a defined data structure. The names of any data structures needed to implement the API are specified by Autonertia to preserve portability, but the implementer of the API is free to declare the elements of the data structure however they choose. This provides the implementer the freedom to design whatever is necessary to meet the requirements of the API, given their particular hardware implementation.

## SCOPE OF DELIVERY

Built and tested Slide-Out Motor, and Controller API as described in this document.

## EXAMPLE INTERFACE FUNCTION TABLE

|  |  |
| --- | --- |
| **Description** | Modifier function to set the duty cycle of the PWM output signal |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_Set\_PWM\_DC** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | | unsigned short int | | HWPWMDutyCycle | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | % | H=E\*327.68 | 0 | 100 | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_SetPWM\_DC ( ) is used to set the duty cycle of the PWM output. The function must  clamp the range of the requested duty cycle value (HWPWMDutyCycle) to the minimum  and maximum specified above. If the PWM output driver is not currently commanded on,  the output should not be turned on. The requested duty cycle should begin no later than  the end of the current period. If the requested duty cycle is less than the current duty  cycle and the current duty cycle has gone farther than the requested duty cycle, the duty  cycle can be terminated. This should alleviate the need to require interrupts at the end of  the period to manage the output. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. Use of this function logically must follow invocations of the function  HW\_InitPWM ( ). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

This function interface chart is generic to Autonertia software architecture documentation. For the purpose of this API, some of the fields may seem unnecessary. In particular, the section on code size and speed is very difficult to determine in this context, as Autonertia is not seeking to provide a clear idea about the HWIO implementation, and it is not always possible to determine what amount of software will be necessary to fulfill the API. The size and speed fields are filled out for some functions when it is the intent of Autonertia to require very fast, in-lined accessors for returning data.

The scaling information is provided for conversion between engineering value (E) and computer value (H).

Note that the interface functions provide only a small portion of the total software the base software group will need to develop. Initialization, processing, diagnostic and powerdown routines will need to be developed to fulfill the API.

**Commonly Used Type Definitions**

C native data types should be used in the implementation of the API.

**Compiler options:**

The following compiler options should be used when supplier software is compiled:

TBD – Supplier Compiler options shall be provided to Autonertia, Inc.

# REFERENCE DOCUMENTS

|  |  |  |  |
| --- | --- | --- | --- |
| FILE NAME | LOCATION | CONTACT | DESCRIPTION |
|  |  | J. Gratz | Motor Controller Schematic |
|  |  | J. Gratz | Controller Schematic |
|  |  |  |  |
|  |  |  |  |

# MEMORY MANAGEMENT

Application software will be responsible for the management of memory. Cache and the memory management unit will be managed by application software. Base software shall initialize and provide access to shared RAM memory for use by the core.

Base software will provide interfaces for NVRAM storage and EEPROM. The option “write to flash” will not be used in this project. External EEPROM will be used.

## BASE SOFTWARE MEMORY

Accommodations are made for supplier to allocate RAM and const data in order to fulfill the API. Application software provides distinct areas for base software to allocate RAM and const data, controlled using the following pragma statements. All data must be allocated specifically to a linker section. Default compiler sections are not allowed.

### VOLATILE RAM

Base software will use the following pragma when allocating volatile RAM:

#pragma section SDATA ".APIvram"

This RAM will exist within the application software’s VRAM section. Application software will be responsible for clearing this RAM to zero before base software needs to or can access it. Note that compiler generated algorithms to initialize this memory is not provided. Initialization of memory will be performed via function call.

### NON-VOLATILE RAM

Base Software will use the following pragma when allocating non-volatile RAM:

#pragma section SDATA ".APInvram"

This RAM will exist within the application software NVRAM section. This section of RAM will be check-summed at powerdown and re-evaluated at the subsequent initialization (after restoring from flash if “write to flash” is used instead of battery backed). In the event that the check-sums do not match, this RAM will be cleared to zero. Note that compiler generated algorithms to initialize this memory is not provided. Initialization of memory will be performed via function call.

### GUARDED NON-VOLATILE RAM

Base software will use the following pragma when allocating guarded non-volatile RAM:

#pragma section SDATA ".APIgnvram"

This RAM will exist within the application software Guarded NVRAM section. This section of RAM will not be check-summed at powerdown nor re-evaluated at the subsequent initialization. This section of RAM will be restored from flash if “write to flash” is used instead of battery backed. Application software will only clear this area to zero in the event of an ECC failure.

### DMA RAM

Base software will use the following pragmas when allocating cache inhibited RAM manipulated by the DMA:

#pragma section SDATA ".APIdmaram\_4"

#pragma section SDATA ".APIdmaram"

This RAM will exist within the application software VRAM section. DMA RAM that requires 4 byte alignment should be allocated in the .APIdmaram\_4 section. Other DMA RAM should be allocated in the .APIdmaram section. Application software will be responsible for clearing this RAM to zero before base software needs to or can access it. Note that compiler generated algorithms to initialize this memory are not provided. Initialization of memory will be performed via function call.

### CONST DATA

Base software will use the following pragma when allocating and initializing const (ROM) data at compile time:

#pragma section SCONST ".APIconst"

This data will exist within the Application software Fixed Calibration section.

### EEPROM DATA

Base software use and responsibilities for EEPROM will be managed through a coordinated

document under Autonertia configuration management detailing exactly which areas of EEPROM storage will be allocated to each party.

### CODE ROM

For the code elements that have no impact on Autonertia level 2 reliability requirements, the

following pragma statement should be used:

#pragma section CODE ".APIcode"

For the code elements that do have an impact on Autonertia level 2 reliability requirements, the following pragma statement should be used:

#pragma section CODE ".APIlv2c"

After review of base software’s implementation, Autonertia will determine which, if any, code elements need to be placed in the .APIlv2c section.

### RAM/ROM USAGE REPORTING

Whenever base software provides updates to the API to Autonertia, documentation indicating the usage in each of the memory areas is required in order to enable Autonertia to properly set up the link.

## NVRAM MANAGEMENT

The controller will implement NVRAM using a battery backed section of the controllers SRAM.

## EEPROM MANAGEMENT

EEPROM memory can be either a physical EEPROM device or emulated if using a “write to

flash” mechanism. Note that if a “write to flash” mechanism is employed, it is expected that

writes will be performed in a timely manner after the software requests data be written to

EEPROM. This differs from NVRAM strategies utilizing a “write to flash” strategy where the

mechanism is not employed until powerdown.

Regardless of the physical mechanism employed, the permanent storage will be “mimicked” into a contiguous block of NVRAM.

The EEPROM space is subdivided into two sections. One section is allocated to Application

software and the other one is allocated to base software. A coordinated document under configuration management will be maintained by Autonertia to identify Application software and base software usage of EEPROM. This document will spell out the exact locations that each organization is using.

Autonertia will be responsible for assuring that only 1 request from Autonertia software to write to EEPROM will be active at any time. Base software will be responsible for maintaining their own writes to EEPROM. Base software is required to update the mimic RAM as well as EEPROM storage for their own EEPROM usage in the same manner as Autonertia’s EEPROM data is managed.

EEPROM management must work seamlessly with the bootloader functions that interface with EEPROM.

### INTERFACE STRUCTURE

The following data structure is employed in managing the EEPROM:

typedef struct

{

unsigned char\* RAMStartAddress;

unsigned short int EEOffset;

unsigned short int TableSize;

unsigned short int CheckSum;

} HWEETableStruct;

Where RAMStartAddress is a pointer to the address in mimic RAM corresponding to the

EEOffset in the EEPROM. TableSize is the number of bytes that should be manipulated.

CheckSum is calculated as 2’s complement of sum of high word of RAMStartAddress, low word of RAMStartAddress, EEOffset, and TableSize. This structure represents a contiguous block of memory that will either be read out of the EEPROM into RAM or written from RAM to the EEPROM.

The base software is encouraged to optimize the underlying implementation of physically storing/reading the data by identifying from the addressing and size information when larger data elements can be written or read to/from physical storage, thereby reducing the total time necessary to complete the transfer. Additionally, in the event that physical storage for some data requested to be written is already equal to the requested data, no physical store needs to be made redundantly.

Before writing to Physical EEPROM the supplier will have to verify the data in HWEETableStruct using checksum. Verification is successful if sum of high word of RAMStartAddress, low word of RAMStartAddress, EEOffset, TableSize and checksum is Zero.

### HW\_InitializeEEMimicRAM ()

|  |  |
| --- | --- |
| **Description** | Function to initialize mimic RAM from EEPROM storage |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitializeEEMimicRAM** | | | | | **Arguments** | **Type** | | **Name** | | | HWEETableStruct \* | | TablePtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | 0 | 1 | | **Semantics:**  0 = HW\_EEPROM\_INIT\_SUCCESS  1 = HW\_EEPROM\_INIT\_FAILED  This function will initialize the RAM specified by TablePtr.RAMStartAddress and  TablePtr.TableSize from the EEPROM storage specified by TablePtr.EEOffset. In the event that the transfer of data fails for any reason, the function will write 0xFF to all bytes specified by the function argument and return the appropriate value to indicate failure. This function is run during initialization and should not return until the transfer is complete or a failure has been identified.  This function should not set any flags in the supplier diagnostic registers specified in  chapter *Controller Specific HW Diagnostics*. Supplier is encouraged to store information  internal to their software for debuggining purposes in the event that a failure to initialize  the mimic RAM occurs. | | | | | | **Usage Restrictions:**  None. | | | | | |
| **Error Handling** | Return value will indicate success/fail. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_InitiateEEPROMWrite ()

|  |  |
| --- | --- |
| **Description** | Function to request storage of RAM to EEPROM |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitializeEEMimicRAM** | | | | | **Arguments** | **Type** | | **Name** | | | HWEETableStruct \* | | TablePtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | 0 | 1 | | **Semantics:**  0 = HW\_EEPROM\_WRITE\_ACCEPTED  1 = HW\_EEPROM\_WRITE\_REJECTED  This function will store the RAM specified by TablePtr.RAMStartAddress and  TablePtr.TableSize into the EEPROM storage specified by TablePtr.EEOffset. In the event that the write request cannot be accepted for any reason, the function will return the appropriate value to indicate failure. | | | | | | **Usage Restrictions:**  Autonertia will coordinate EEPROM write priority and is responsible for assuring that only  one request is actively requested at any one time. Autonertia will use the function  HW\_EEWritesAreComplete(), specified below, to identify when a new write can be  requested. | | | | | |
| **Error Handling** | Return value will indicate whether the function accepts the write request or not. It is  understood that writes to EEPROM takes additional time and will be done in background  processes over several time loops. In the event of a failure in the background  processing, mechanisms for reporting are outlined in the chapter *Controller Specific HW*  *Diagnostics*. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EEWritesAreComplete ()

|  |  |
| --- | --- |
| **Description** | Accessor function to identify when a new EEPROM write can be requested |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EEWritesAreComplete** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | 0 | 2 | | **Semantics:**  0 = HW\_EEPROM\_WRITES\_PENDING  1 = HW\_EEPROM\_WRITES\_COMPLETE\_NO\_FAILURE  2 = HW\_EEPROM\_WRITES\_COMPLETE\_FAILED  This function will identify when supplier software has completed previous Autonertia  EEPROM write request (whether write was successful or not) and can accept additional  write requests. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | This function will identify when supplier software has completed previous Autonertia  EEPROM write requests and can accept additional write requests.  If no write request has been made before calling this function, supplier will return  HW\_EEPROM\_WRITES\_COMPLETE\_NO\_FAILURE.  If the write operation is still in progress, supplier will return  HW\_EEPROM\_WRITES\_PENDING.  If the previous write operation is complete with no failure, supplier will return  HW\_EEPROM\_WRITES\_COMPLETE\_NO\_FAILURE.  If the previous write operation is complete but failed, supplier will return  HW\_EEPROM\_WRITES\_COMPLETE\_FAILED. Subsequent calls to  HW\_EEWritesAreComplete() should return HW\_EEPROM\_WRITES\_COMPLETE\_NO\_FAILURE until a new write request has been  received.  Supplier will not report any EEPROM writing failure in the diagnostic registers listed in  the chapter *Controller Specific HW Diagnostics*. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation.

For any const data required in the implementation, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

For any RAM needed to fulfill the API, the following pragma statements should be used when declaring the RAM, depending upon the storage class required:

#pragma section SDATA ".APIvram"

#pragma section SDATA ".APInvram"

#pragma section SDATA ".APIgnvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that VRAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear VRAM to zero.

For the code elements that have no impact on Autonertia level 2 requirements, the following

pragma statement should be used:

#pragma section CODE ".APIcode"

For the code elements that do have an impact on Autonertia level 2 requirements, the following pragma statement should be used:

#pragma section CODE ".APIlv2c"

After reviewing supplier’s implementation, Autonertia will determine which, if any, code elements need to be placed in the .APIlv2c section.

### HEADER FILES

Supplier will also provide an associated header file containing external references to any

required data and any function prototypes needed. The file naming convention for interface files is defined in the chapter on “Header File Naming Convention”. At the implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Whenever supplier provides updates to the API to Autonertia, documentation indicating the usage in each of the memory areas is required in order to enable Autonertia to properly set up the link.

# INTERRUPTS & EXCEPTIONS

Interrupts and CPU Exceptions will be managed entirely by Autonertia. It is understood however, that the supplier of the controller will have strategies that require the use of interrupts and these will be accommodated according to the criteria outlined below. Additionally, supplier will generate certain interrupts, as outlined below, to enable Autonertia to implement their control strategies. Supplier will need to implement software to generate these interrupts as they are contingent upon how the microcontroller pins are allocated.

Supplier shall not use a CPU exception to fulfill a control or operating strategy. All supplier

interrupts shall be accomplished via the External Interrupt Handler of the Microcontroller.

Autonertia is implementing the Software Vector Mode of the Microcontroller Interrupt Controller. It is supplier’s responsibility to clear the source of the interrupt before completing the function call handling that interrupt.

### MINIMUM INTERRUPT REQUIREMENTS

|  |  |  |
| --- | --- | --- |
| **Interrupt** | **Reserved Source** | **Priority** |
| Motor Control Global Exception | Timer Handling Motor Phase PWM | 15 (Highest) |
| Operating System Kernels | PIT Timer 0 - 1 | 14 |
| Invocation of tasks | Software Settable  Interrupts 1 - 8 | 5 |
|  |  |  |
|  |  |  |

The priorities in the above table are not fixed, but indicate a relative priority that must be

maintained. Priority definitions will be established based on coordinated Autonertia interrupt

requirements and supplier interrupt requirements. Supplier should provide interrupt

requirements to Autonertia as soon as possible.

## IRQ PIN INTERRUPTS – EDGE DETECTS

The Edge Detects API will consist of a generic interface common to all hardware

implementations of IRQ pins. A common data structure instantiated once for each IRQ pin will be used to manage the interface.

### INTERFACE STRUCTURE

Each IRQ pin interfaced should have an instantiated structure type defined as HWEdgeDetect (to be provided by Supplier). A separate instance of this structure will be created for each IRQ pin to be interfaced. This data structure will be used in the implementation to manage each IRQ pin.

The following type definition is used to select which edge(s) to identify activity on.

typedef enum

{

HW\_DETECT\_ON\_FALLING\_EDGE,

HW\_DETECT\_ON\_RISING\_EDGE,

HW\_DETECT\_ON\_BOTH\_EDGES

} HWEdgeDetectLevelType;

### HW\_EdgeDetectInit ()

|  |  |
| --- | --- |
| **Description** | Initializes the edge detect for operation. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectInit** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | | HWEdgeDetectLevelType | | EdgeLevel | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | n/a | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will set up the IRQ pin to detect edges as specified by EdgeLevel, disable  the interrupt, and clear the associated interrupt status flag. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EdgeWasDetected ()

|  |  |
| --- | --- |
| **Description** | Indentifies if an edge has been detected. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectInit** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if an edge (specified in  HW\_EdgeDetectInit() ) has been identified on the IRQ pin. The value returned  should reflect the actual status flag from the interrupt status register. The function should  clear the status flag before returning. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: 36 bytes | | Speed: In-lined | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EdgeDetectPinIsLow ()

|  |  |
| --- | --- |
| **Description** | Accessor function to identify if the IRQ pin is being driven low. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a flag indicating if the signal on the IRQ pin is low. A non-zero  value will indicate that the IRQ pin is low. A zero will be returned if IRQ pin is high. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: 36 bytes | | Speed: In-lined | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EdgeDetectPinIsHigh ()

|  |  |
| --- | --- |
| **Description** | Accessor function to identify if the IRQ pin is being driven high. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsHigh** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a flag indicating if the signal on the IRQ pin is high. A non-zero  value will indicate that the IRQ pin is high. A zero will be returned if IRQ pin is high. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: 36 bytes | | Speed: In-lined | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EdgeDetectInterruptEnable ()

|  |  |
| --- | --- |
| **Description** | Enables the interrupt associated with the IRQ pin |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function enables the interrupt associated with the IRQ pin. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: 24 bytes | | Speed: In-lined | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_EdgeDetectInterruptDisable ()

|  |  |
| --- | --- |
| **Description** | Disables the interrupt associated with the IRQ pin |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | HWEdgeDetect \* | | EdgeDetectPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function disables the interrupt associated with the IRQ pin. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: 24 bytes | | Speed: In-lined | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## POWER SUPPLY LOW VOLTAGE DETECTION

A strategy needs to be implemented to protect for low battery voltage situations, i.e., when the voltage drops below 5.0 V but remains above 3.3 V. It is expected that this will occur only during cold start cranking scenarios, for a maximum of 50 µs. As the module cannot be reset during this condition, actions must be taken to protect data integrity of A2D conversions, monitor possible false signals on sensors, and reset/re-initialize any devices on the board that may be vulnerable to battery voltages below 5 volts.

To implement this strategy, it will be necessary to connect the 5V reset pin from the power

supply to an IRQ pin on the Microcontroller. The edge detect API will be implemented on the IRQ pin to provide Autonertia a means of monitoring for and responding to the event.

For Autonertia’s purposes, it will be necessary to interrupt on the falling edge of the 5V reset pin. An interfacing function, HW\_PSUndervoltageFallingEdgeHandler(), is provided below for supplier to implement a strategy on the falling edge of the interrupt. Although not specifically needed by Autonertia, the interrupt can be configured to trigger on the rising edge as well, if supplier’s strategy requires it. An interfacing function,

HW\_PSUndervoltageRisingEdgeHandler(), is provided below for supplier to implement a

hardware strategy on the rising edge of the interrupt.

Due to the nature of this interrupt, it is given a very high priority, so every effort should be made to create a fast response during the interrupt.

### HW\_PSUndervoltageFallingEdgeHandler()

|  |  |
| --- | --- |
| **Description** | Function for supplier to implement a low battery voltage protection strategy in, on the  falling edge of the 5V reset pin. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will perform supplier’s low battery voltage protection strategy, on the falling  edge of the 5V reset pin. | | | | | | **Usage Restrictions:**  Only to be called in the interrupt associated with the 5V reset pin of the power supply  and only on the falling edge of that interrupt. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

### HW\_PSUndervoltageRisingEdgeHandler()

|  |  |
| --- | --- |
| **Description** | Function for supplier to implement a low battery voltage protection strategy in, on the  rising edge of the 5V reset pin. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will perform supplier’s low battery voltage protection strategy, on the rising  edge of the 5V reset pin. | | | | | | **Usage Restrictions:**  Only to be called in the interrupt associated with the 5V reset pin of the power supply  and only on the rising edge of that interrupt. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## ENABLING AND DISABLING INTERRUPTS

It is understood that supplier code may need to disable interrupts to protect critical areas of

code. Supplier must be certain to restore the interrupt state correctly. As it is not Autonertia’s

intent to provide interfacing to the supplier, the following sample code is provided to supplier. Both organizations should assure implementation in same method. Supplier will have to share the implementation information with Autonertia.

The intent of the following sample code is to create a 32-bit local variable and use its address as the argument to the function:

asm void DisableInterruptsV(unsigned long \* SavedMSR)

{

% reg SavedMSR;

! "r4"

mfmsr r4 // store the value of MSR in r4

stwx r4, 0, SavedMSR // store r4 (value of MSR) in

// SavedMSR

wrteei 0 // write to MSR[EE] immediate value of 0

// (clear MSR[EE])

}

asm void ReEnableInterruptsV(unsigned long \* SavedMSR)

{

% reg SavedMSR;

! "r4"

lwzx r4, 0, SavedMSR // Get the data stored in SavedMSR and

// store in r4

wrtee r4 // restore MSR[EE] from r4 w/o altering

// other MSR bits that may have changed.

// (page 3-391 of EREF)

}

These two function calls must wrap the critical code area. The use of interrupts during

initialization routines must be coordinated with Autonertia prior to releasing software.

Disabling interrupts is closely monitored in Autonertia software systems. Any supplier software that will disable interrupts must be reviewed by Autonertia before being integrated into Autonertia

software. Autonertia will need to know the frequency and maximum duration of each call to

disable interrupts, the software components that are disabling the interrupts, and whether there are alternatives available to prevent disabling interrupts and what impact those alternatives might have.

## WATCHDOG SERVICING

It is understood that supplier software will need to service the watchdog to prevent a reset

during longer processes that may run during initialization and powerdown. Autonertia will be

responsible for servicing the watchdog when the software is not in the initialization and

powerdown states. The watchdog will be set to trigger a reset or interrupt if not serviced within 55 ms. As it is not Autonertia’s intent to provide interfacing to the supplier, the following code is provided to supplier to assure that both organizations implement the same method.

void ServiceWatchdogV()

{

asm(" lis r3,0xC000");

// Clear ENW & WIS

asm(" mtspr TSR,r3"); // Write bit to clear bits

}

Servicing the watchdog is closely monitored in Autonertia software systems. Any supplier

software that will service the watchdog must be reviewed by Autonertia before being integrated into Autonertia software. Autonertia will need to know which supplier functions are servicing the

watchdog and how long those functions will run before returning.

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation. The

instantiated data structure for the IRQ pin will be named EdgeDetect01.

The instantiated edge detect structure should be implemented as const data. For all const data needed to fulfill the API, the following pragma statement should be used when declaring the const data:

#pragma section SCONST ".APIconst"

For any RAM needed to fulfill the API, the following pragma statement should be used when declaring the RAM:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of the

instantiated structure and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header. Where

specifically noted, implementer is expected to use in-lined functions.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required indicating which devices are used to implement interrupts on so that Autonertia can correctly set interrupt priorities and populate the vector table. For supplier

interrupts in addition to those specifically requested by Autonertia, supplier must also provide the function names for those interrupt handlers as well. Supplier must provide execution time estimates for all supplier interrupts expected to be implemented in the system. Lastly, supplier must provide documentation on all algorithms that disable interrupts as described above in section 5.4.

Documentation that explains supplier’s power supply low voltage strategy shall also be

provided. This documentation must include timing estimates for the functions

HW\_PSUndervoltageFallingEdgeHandler() and HW\_PSUndervoltageRisingEdgeHandler() if implemented.

# CONTROLLED RESET

At various times, it is necessary to perform a controlled reset of the entire controller. Autonertia software performs this controlled reset when CPU exceptions are encountered, when functional safety fails, and when the module is in powerdown and an ignition key request to re-start the module occurs or when LIN flashing has been requested.

When the controlled reset is requested, the entire controller must be reset. Any devices on the board that should be reset must be accommodated when the reset function is invoked. The reset function should never return and should continue to attempt to reset the controller

indefinitely.

Only Autonertia software should invoke this function.

## HW\_SystemReset ()

|  |  |
| --- | --- |
| **Description** | Function to reset the entire controller |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function must be in-lined. No function calls will be made from within this function.  This function performs whatever actions are necessary to reset the entire controller, and  all attached devices that influence/control I/O. The function should never return and  should attempt to reset the controller repeatedly. It is intended that this function takes as  direct a path as possible to resetting the controller. | | | | | | **Usage Restrictions:**  Invoked only by Autonertia software. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: Hardware Dependent | | Speed: In-lined and < 100us | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_SoftReset ()

|  |  |
| --- | --- |
| **Description** | Function to request controller soft reset. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_EdgeDetectPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function must be in-lined. No function calls will be made from within this function.  The function should never return and should attempt to reset the controller repeatedly if  the conditions and execution path to invoke this API is met. It is intended that this  function takes as direct a path as possible to resetting the controller. | | | | | | **Usage Restrictions:**  Invoked only by Autonertia software. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: Hardware Dependent | | Speed: none | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation.

Any const data that needs to be declared should be built with the following pragma statement:

#pragma section SCONST ".APIconst"

Any RAM needed to fulfill the API be built with the following pragma statement:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of the

instantiated structure and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation explaining physical reset capability and how the software invokes the reset on the hardware is required.

# ANALOG TO DIGITAL CONVERSIONS

The interface of analog to digital conversion hardware will be implemented by Supplier. It is

required that the supplier provide documentation regarding the complete hardware mapping of analog signals to ADC channel numbers. It is advised to supplier to make full use of the DMA channels assigned to the A/D module. An agreement regarding the appropriate assignment of DMA channels and their priority levels must be made between Autonertia and the supplier. This topic is documented in the chapter on Direct Memory Access. Supplier has to perform ADC calibration during start-up and periodically during run time.

Different analog signals like power supply voltage values, phase current values, Temperature sensor readings, etc. can be captured through the

ADC port. These values can then be multiplied with respective Engineering scale values to use them in the application code. Based on the trigger mode, all analog signals are divided into two groups, SW and HW triggered.

## SW TRIGGERED A2D

The A2D channel number should be defined and provided by Supplier software in the following way. The example shown has 3 SW triggered signals followed by total number of signals. The actual analog signals will be determined by specific hardware design.

typedef enum

{

PARK\_PAWL\_POS\_SENSE, // Park Pawl Position

TRANS\_TEMP\_SENSE, // Transmission temperature

BATT\_COOL\_TEMP\_SENSE, // Battery coolant temperature

Num\_Of\_A2D\_Channels

} A2DChannelType;

Autonertia software will trigger the analog to digital conversion. Supplier software needs to sample and convert all channels and make the conversion results available.

The following enumeration type (to be provided in supplier’s header) should be used to specify the A2D conversion status:

typedef enum

{

Completed = 0,

Not\_Completed,

Failed

} ADC\_Status;

The following structure type (to be provided in supplier’s header) should be used to return the conversion result to Autonertia’s application software:

typedef struct

{

A2DChannelType channel;

unsigned short result;

} A2DResultType;

Supplier should specify the ADC conversion time in delivered documentation so Autonertia

software can trigger the appropriate ADC conversion before the ADC raw results are read by Autonertia software.

Autonertia expects supplier to use maximum available resolution for analog to digital conversion. Supplier should specify this resolution in the delivered documentation so that Autonertia can use this information when ADC results are read by Autonertia software.

## HW\_InitA2D ()

|  |  |
| --- | --- |
| **Description** | Function to initialize all analog to digital conversion channels, including both SW and HW  triggered channels. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitA2D** | | | | | **Arguments** | **Type** | | **Name** | | |  | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | flag | n/a | 0 - Failed | 1 - Success | | **Semantics:**  This function performs the initialization of all used channels for the ADC module,  including both SW and HW triggered channels. | | | | | | **Usage Restrictions:**  Invoked only by Autonertia software. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_GetRawA2D ()

|  |  |
| --- | --- |
| **Description** | Allows Supplier software to return analog-to-digital conversions used by Autonertia  software |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_GetRawA2D** | | | | | **Arguments** | **Type** | | **Name** | | | A2DResultType \* | | A2DResultPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will update the analog-to-digital conversion of analog input required by  Autonertia software. It is implied that supplier will store the value in RAM for its own  purposes. This function will be invoked during the temporal loop requested by Autonertia. Autonertia expects Supplier to use maximum available resolution for analog to  digital conversion. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_A2D\_Trigger ()

|  |  |
| --- | --- |
| **Description** | Allows Supplier software to return analog-to-digital conversions used by Autonertia  software |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_A2D\_Trigger** | | | | | **Arguments** | **Type** | | **Name** | | |  | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will trigger the analog-to-digital conversion of all channels. This function  needs to be as short as possible as it will be invoked by Autonertia Operating System  kernel. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: As fast as possible | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_A2D\_Completed ()

|  |  |
| --- | --- |
| **Description** | Allows Autonertia software to query ADC status to determine if the specified A/D  conversion is completed |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_A2D\_Completed** | | | | | **Arguments** | **Type** | | **Name** | | |  | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | ADC\_Status | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will return the A2D conversion status. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_A2D\_GetConfigScale ()

|  |  |
| --- | --- |
| **Description** | This function returns the ADC conversion scale used to convert the counts into  engineering value |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_A2D\_GetConfigScale** | | | | | **Arguments** | **Type** | | **Name** | | | A2DChannelType | | A2DChannel | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | ADC\_Status | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function returns a scaling value used to convert the counts to engineering value. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_A2D\_RequestCalibration ()

|  |  |
| --- | --- |
| **Description** | This function is used to request/cancel a calibration for all the A2D Converters during run  time |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_A2D\_RequestCalibration** | | | | | **Arguments** | **Type** | | **Name** | | | unsigned char | | CalibReq | | | unsigned char | | AverageType | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | n/a | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function shall request or cancel a calibration of all the A2D Converters (SW-  triggered and HW-triggered) according to the parameter CalibReq:  0: Cancel the ongoing calibration  1: Request a new runtime calibration.  When new calibration is requested, it shall be performed with the average type specified  by the parameter AverageType based on the samples of the reference voltages to  calibrate the A2D converters:  0: Mode Average (Find the most frequent sample)  1: Mean Average (Accumulate the samples and divide by N)  The calibration shall be started from the end of the next HW-triggered conversion.  In the case a new calibration is requested while another calibration is already in  progress, a new calibration shall be scheduled for the end of the current one. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation.

Any const data that needs to be declared should be built with the following pragma statement:

#pragma section SCONST ".APIconst"

Any RAM needed to fulfill the API be built with the following pragma statement:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will provide a header file containing in-lined functions for accessors to A2D conversions that supplier requires. The file naming convention for interface files is defined in the chapter on “Header File Naming Convention”.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Supplier shall provide documentation identifying the complete hardware mapping of analog

signals to ADC channel numbers. In the event that supplier requires additional A2D’s to be

converted, supplier will provide documentation as outlined above.

# DIGITAL INPUTS

The Digital Inputs interface will be used for all external switched inputs connected to the

controller as well as numerous internal signals inside of the controller. The supplier will be

required to provide the pin state (high/low) for each of these upon request via function calls

defined below. Autonertia will sort out all other aspects of the digital input, e.g., polarity,

multiplexed inputs, on and off times, etc.

In general, Autonertia will request digital input pin states to be updated at a particular rate.

Additionally, these pin states will need to be updated numerous times on demand during module initialization and during powerdown, when the operating system is not running temporal tasks. In the event that “batch” process digital inputs cannot be read due to an identified failure of the device, pin states should be returned as low, or zero, in the returned flag register. Reporting failures for board component hardware is covered in the chapter Controller Specific HW Diagnostics.

## INTERFACE STRUCTURE

One function call will be required to provide access to this data. The function call will return a 32 bit value, which can therefore represent up to 32 digital input pin states.

Additionally another function is required to obtain the instantaneous unfiltered state for each

individual digital input.

An enum DigitalInputType will be used to represent and read each digital input status.

## HW\_UpdateDigInPinState ()

|  |  |
| --- | --- |
| **Description** | Provides access to the digital input pin states |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_UpdateDigInPinState** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will provide the pin states on those inputs defined for this function. Note that  these must be read upon request through this function call. A high pin state should be  reflected as a bit set in the returned flag register. Any bit that does not correspond to a  digital input should be cleared. In the event that “batch” process digital inputs cannot be  read, pin states should be returned as bit clear in the returned flag register. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_GetDigInPin\_InstState ()

|  |  |
| --- | --- |
| **Description** | Provides the instantaneous state for a digital input |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_GetDigInPin\_InstState** | | | | | **Arguments** | **Type** | | **Name** | | | DigitalInputType | | DigitalInputTypeNo | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will provide the instantaneous unfiltered state for the desired input. A high  pin state should be reflected as 1 and low pin state should be reflected as 0. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation.

Any const data that needs to be declared should be built with the following pragma statement:

#pragma section SCONST ".APIconst"

Any RAM needed to fulfill the API be built with the following pragma statement:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will provide a header file containing in-lined functions for accessors to A2D conversions that supplier requires. The file naming convention for interface files is defined in the chapter on “Header File Naming Convention”. At the implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required identifying which bits in the return value of each function correspond to which physical inputs. This includes signal name, schematic name at the controller connector and schematic name at microcontroller for each bit.

# DIGITAL OUTPUTS

This interface software will consist of a generic interface common to all hardware

implementations of digital output devices. A common data structure instantiated once for each output signal will be used to manage the interface. The interface will provide functionality giving higher level software the ability to manipulate the state of each output and to determine the failure status of the device being driven by a given output pin (where applicable). All functionality provided by the interface will remain transparent to higher level software.

Note that high side digital outputs and low side digital outputs will be interfaced in the same

way. It is implied that “turning on” means driving high side outputs high and low side outputs

low. This interface is also used for signals that are internal to the controller. In these cases,

“turning on” means driving the pin high.

It is recognized that low side outputs are often processed temporally as a collection (such as a SPI output device) and actual activation of those signals will be deferred until such time as that temporal process is invoked. While that will work fine when temporal tasks are running, there are multiple times during the module initialization and powerdown when it is necessary to immediately process these output requests. As such, supplier must provide a function call for Autonertia to invoke that will process these output requests on demand. The processing function cannot return control to Autonertia code until the outputs have been set. All other signals are expected to be manipulated immediately.

## INTERFACE STRUCTURE

A data structure with type HWDigOutput will be defined by Supplier. A separate instance of this structure will be created for each digital output signal. This data structure will be used in the implementation to manage each output device.

For diagnostic purposes, a return value composed of 1 or more values from the following

enumeration is required:

typedef enum

{

HW\_NO\_FAILURE\_PRESENT = 0x00,

HW\_FAILURE\_PRESENT = 0x01,

HW\_SHORTED\_LOW = 0x02,

HW\_SHORTED\_HIGH = 0x04,

HW\_OPEN\_CIRCUIT = 0x08,

HW\_DRIVER\_OVERTEMP = 0x10,

HW\_NOT\_DIAG\_AT\_THIS\_TIME = 0x40,

HW\_NOT\_DIAG\_CAPABLE = 0x80,

HW\_DRIVER\_OVERVOLTAGE = 0X100,

HW\_DRIVER\_UNDERVOLTAGE = 0X200}

HWDigitalOutputFailureType;

If there is no failure identified, the return value should be HW\_NO\_FAILURE\_PRESENT. If there is a failure identified then the value should be HW\_FAILURE\_PRESENT, at a minimum. If it is possible to detect additional information, then the other flags should be logically OR’d to the return value. If the output is not diagnosable, then HW\_NOT\_DIAG\_CAPABLE should be returned.

It is understood that “batch” processing of outputs diagnostics may be performed at some

temporal rate. While that will work fine when temporal tasks are running, there are multiple

times during the module initialization when it is necessary to immediately refresh these output diagnostics. As such, supplier must provide a function call for Autonertia to invoke that will refresh these output diagnostics on demand. The processing function cannot return control to Autonertia code until the failure information for each output has been determined.

## HW\_TurnOnDigOut ()

|  |  |
| --- | --- |
| **Description** | Turn the specified digital output signal “ON” |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_TurnOnDigOut** | | | | | **Arguments** | **Type** | | **Name** | | | HWDigOutput \* | | HWDigOutPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  Drives the pin associated with this particular signal to the “ON” state. | | | | | | **Usage Restrictions:**  Devices must be initialized prior to calling these functions. It is understood that “batch”  processing of certain low side outputs will defer actuation of certain outputs. While that  will work fine when temporal tasks are running, there are multiple times during the  module initialization and powerdown when it is necessary to immediately process these  output requests. As such, supplier must provide a function call for Autonertia to invoke that  will process these output requests on demand. The processing function cannot return  control to Autonertia code until the outputs have been set. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_TurnOffDigOut ()

|  |  |
| --- | --- |
| **Description** | Turn the specified digital output signal “OFF” |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_TurnOffDigOut** | | | | | **Arguments** | **Type** | | **Name** | | | HWDigOutput \* | | HWDigOutPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  Drives the pin associated with this particular signal to the “OFF” state. | | | | | | **Usage Restrictions:**  Devices must be initialized prior to calling these functions. It is understood that “batch”  processing of certain low side outputs will defer actuation of certain outputs. While that  will work fine when temporal tasks are running, there are multiple times during the  module initialization and powerdown when it is necessary to immediately process these  output requests. As such, supplier must provide a function call for Autonertia to invoke that  will process these output requests on demand. The processing function cannot return  control to Autonertia code until the outputs have been set. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_DigOutIsFailing()

|  |  |
| --- | --- |
| **Description** | Indicates if the requested digital output is currently in a failing state |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_DigOutIsFailing** | | | | | **Arguments** | **Type** | | **Name** | | | HWDigOutput \* | | HWDigOutPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | HWDigitalOutputFailureType | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flags | n/a | n/a | n/a | | **Semantics:**  For the digital output identified by HWDigOutPtr, returns a value made up from one or  more of the values in the enumeration identified above in Section 9.1. Note that the  implementer is responsible for any software necessary to reset output drivers (such as  driving the output to the opposite state) in the event that a failing condition is identified. If  this algorithm is intrusive to the commanded output state, the intrusion must be  minimized. The implementer must be certain that the device can be evaluated anew at a  particular rate, when the temporal tasks are running. Additionally, it is necessary to  evaluate possible failure states numerous times during initialization, before temporal  tasks are running. As such, supplier must provide a function call for Autonertia to invoke  that will refresh failure information on demand. The refreshing function cannot return  control to Autonertia code until the failure information for each digital output has been  determined. This function call can be combined with the similar function call for PWM  outputs. | | | | | | **Usage Restrictions:**  Only applicable for digital outputs that have a diagnostic capability. Generally, signals  that are internal to the controller do not have diagnostic capability. In this case, the  function will not normally be called but, in any event, an enumerated value is defined for  the event when this function is called for signals that have no diagnostic capability. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing an instantiated data structure of type HWDigOutput for each digital output device, the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation. The instantiated data structures will adopt the following naming convention:

HWDigitalOutput01

HWDigitalOutput02

.

.

HWDigitalOutputxx

The instantiated structures should be implemented as const data whenever possible. To

properly locate this data, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

If the structures (or any subset of them) need to be implemented in RAM, or for any other RAM needed to fulfill the API, the following pragma statement should be used when declaring the RAM:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that this RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear this RAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required indicating which HWDigOutput instance is to be used for which

actual output. Documentation explaining the diagnostic methodology for each output and any physical output driver shall also be provided. Timing diagrams illustrating any intrusive

diagnostic methodologies must be provided.

# PWM OUTPUTS

The Pulse Width Modulation API will have a unique data structure type and set of interface

functions. A data structure instantiated once for each output will be used to manage the

interfaces. Section 10.1 is dedicated to Autonertia PWM output requirements are covered in section 10.2.

### Autonertia PWM Interface Structure

Each PWM device for should have an instantiated structure type defined as HWPWMOut or

HWHSPWMOut (if High Speed PWM device is used). A separate instance of these structures will be created for each hardware device. These data structures will be used in the implementation to manage each PWM output device.

For PWM diagnostic purposes, a function return value composed of 1 or more values from the following enumeration is required:

typedef enum

{

HW\_NO\_FAILURE\_PRESENT = 0x00,

HW\_FAILURE\_PRESENT = 0x01,

HW\_SHORTED\_LOW = 0x02,

HW\_SHORTED\_HIGH = 0x04,

HW\_OPEN\_CIRCUIT = 0x08,

HW\_DRIVER\_OVERTEMP = 0x10,

HW\_NOT\_DIAG\_AT\_THIS\_TIME = 0x40,

HW\_NOT\_DIAG\_CAPABLE = 0x80,

HW\_DRIVER\_OVERVOLTAGE = 0X100,

HW\_DRIVER\_UNDERVOLTAGE = 0X200

} HWPWMOutputFailureType;

If there is no failure identified, the return value should be HW\_NO\_FAILURE\_PRESENT. If there is a failure identified then the value should be HW\_FAILURE\_PRESENT, at a minimum. If it is possible to detect additional information, then the other flags should be logically OR’d to the return value. If the output is not diagnosable, then HW\_NOT\_DIAG\_CAPABLE should be returned.

It is understood that “batch” processing of outputs diagnostics may be performed at some

temporal rate. While that will work fine when temporal tasks are running, there are multiple

times during the module initialization when it is necessary to immediately refresh these output diagnostics. As such, supplier must provide a function call for Autonertia to invoke that will refresh these output diagnostics on demand. The processing function cannot return control to Autonertia code until the failure information for each PWM output has been determined.

Output signal polarity is controlled using the following enumerated type:

typedef enum

{

HW\_LOGIC\_LOW\_DRIVEN,

HW\_LOGIC\_HIGH\_DRIVEN

} HWPolarityType;

To clarify, if the polarity for an output is identified as HW\_LOGIC\_HIGH\_DRIVEN, the duty cycle percentage specified in the function HW\_SetPWM\_DC() should be driven high.

### 3 Phase Inverter Driver with Dead Time Insertion

#### Counter Configuration

* Dead time insertion/generation for PWM signal must be possible using micro-processor based dead-time insertion.
* The new computed Task0 time period (for variable frequency control for the reduction of inverter switching losses) value provided by application needs to be loaded on fly in cycle (n) into the load register of the counter so that counter generates this new time period in next cycle (n+1).

#### Task0 PWM Cycle Period

* The time period of Task0 PWM cycle is computed by the Application for the variable frequency current control and is then passed to the low level SW. The PWM cycle period event shall generate a hardware trigger to start the ADC. The DMA Unit shall be used for conversion results transfer to increase software throughput efficiency. Once all conversions for these signals have been completed, including DMA transfer, an interrupt is generated to start PWM Task0. Thus, actual Task0 period shall be referred to as the time interval between two consecutive events of completion of DMA’s transferring ADC results onto internal ram variables.
* Refer to Section 7 for the signals to be included in each of the ADC acquisition sets.
* Refer to the ‘PWM Waveform Requirements’ diagram below for an illustration showing the completion of the hardware triggered A/D acquisition prior to starting PWM Task0.

#### Phase A, B and C PWM

* The PWM period and duty cycle of Phase A, B and C PWM signal is computed by the Application.
* These 6 Phase PWM needs to be center – aligned across the Task0 time period.
* The polarities of default inactive PWM output signals are recommended to be “digital high (H)” level. But notice that the timing chart following shows low level as inactive PWM outputs for easier understandings.

#### Usage of API

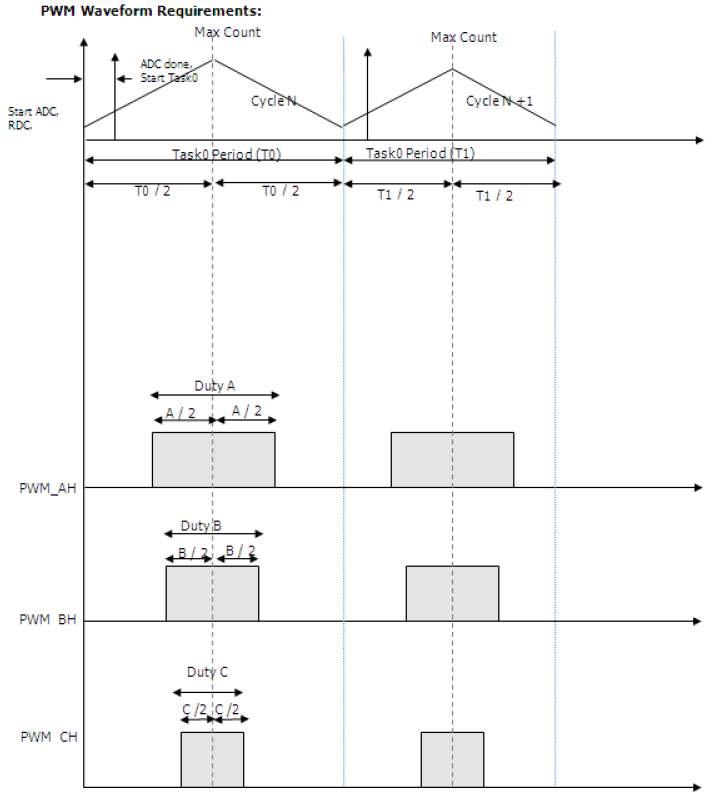
The following four (4) changes will be required of this existing API:

1. A/D Sampling – Refer to Section 11.2 for requirements regarding 4 channel simultaneous sampling and the sets of signals to be sampled.

2. Dead-time sw configurable via dynamic parameter (not hard-coded constant).

3. Change to “…Get” function --- faults to be returned are not just the high-priority fault, as currently exists in the API, but rather return all active faults. Require n-bit words with each bit representing a particular possible fault. Require duplicate n-bit words, but with only one bit set, representing the first fault which was set (in the event there were multiple bits set).

4. Require API to “Set PWM Output To Null Vector”, meaning All Switches Open or 3 Phase Short.



## HW\_InitPWM ()

|  |  |
| --- | --- |
| **Description** | Indicates if the requested digital output is currently in a failing state |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitPWM** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | | HWPolarityType | | HWPWMPol | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | n/a | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_InitPWM ( ) initializes the PWM output driver pointed at by HWPWMPtr. If the  Polarity is equal to HW\_LOGIC\_HIGH\_DRIVEN (“OFF” state is low) the initial state of  the PWM output will be set low. If the Polarity is equal to HW\_LOGIC\_LOW\_DRIVEN  (“OFF” state is high) the initial state of the PWM output will be set high. The PWM output  device will be set to a 0 Hz frequency and 0% duty cycle so that if the function  HW\_TurnOnPWM() is invoked without setting the frequency and duty cycle, the output  will remain at its inactive state. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_SetPWMFreq ()

|  |  |
| --- | --- |
| **Description** | Modifier function to set the frequency of the PWM output signal |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_SetPWMFreq** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | | unsigned short int | | HWPWMFreq | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | Hz | H=E | 0 | 2000 | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_SetPWMFreq ( ) is used to set the frequency of the PWM output. The function must  clamp the range of the requested frequency value (HWPWMFreq) to the minimum and  maximum specified above. If the PWM output driver is not currently on, the output  should not be turned on. The requested frequency change should take effect no later  than the end of the current period. If a non-zero duty cycle is already indicated, that duty  cycle must be maintained at the new frequency. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. Use of this function logically must follow invocations of the function  HW\_InitPWM ( ). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_SetPWM\_DC ( )

|  |  |
| --- | --- |
| **Description** | Modifier function to set the duty cycle of the PWM output signal |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_SetPWM\_DC** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | | unsigned short int | | HWPWMDutyCycle | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | % | H=E\*327.68 | 0 | 100 | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_SetPWM\_DC ( ) is used to set the duty cycle of the PWM output. The function must  clamp the range of the requested duty cycle value (HWPWMDutyCycle) to the minimum  and maximum specified above. If the PWM output driver is not currently commanded on,  the output should not be turned on. The requested duty cycle should begin no later than  the end of the current period. If the requested duty cycle is less than the current duty  cycle and the current duty cycle has gone farther than the requested duty cycle, the duty  cycle can be terminated. This should alleviate the need to require interrupts at the end of  the period to manage the output. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. Use of this function logically must follow invocations of the function  HW\_InitPWM ( ). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_TurnOnPWM()

|  |  |
| --- | --- |
| **Description** | Function to initiate the driving of the PWM output. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_TurnOnPWM** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_TurnOnPWM( ) turns the PWM output driver on. The output signal will operate  with the last requested frequency and duty cycle as described in the sections above. If  the output is already on, the driver must prevent a glitch from occurring in the signal. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. Use of this function logically must follow invocations of the function  HW\_InitPWM ( ). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_TurnOffPWM()

|  |  |
| --- | --- |
| **Description** | Function to turn off the PWM signal |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_TurnOffPWM** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_TurnOffPWM() turns the PWM output driver off immediately (no waiting until end of  current period). The pin state for “OFF” is defined as the opposite of the polarity setting  last made for the object, whether from the object’s initialization HW\_InitPWM() or from a  call to HW\_SetPWMPolarity(). The output driver must maintain knowledge of  previously requested frequency and duty cycle so that a subsequent call to  HW\_TurnOnPWM() will correctly drive the device. | | | | | | **Usage Restrictions:**  Devices upon which the PWM objects are implemented must be initialized and ready to  accept this function. Use of this function logically must follow invocations of the function  HW\_InitPWM ( ). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PWMIsFailing ( )

|  |  |
| --- | --- |
| **Description** | Accessor function to identify is a PWM signal is in a failing state. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PWMIsFailing** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | HWPWMOutputFailureType | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flags | n/a | n/a | n/a | | **Semantics:**  For the PWM output identified by HWPWMPtr, returns a value made up from one or more  of the values in the enumeration identified above in section 10.1. Note that the  implementer is responsible for any software necessary to reset output drivers (such as  driving the output to the opposite state) in the event that a failing condition is identified. If  this algorithm is intrusive to the commanded output state, the intrusion must be  minimized. The implementer must be certain that the device can be evaluated anew  every 12.5 msec when the temporal tasks are running. Additionally, it is necessary to  evaluate possible failure states numerous times during initialization, before temporal  tasks are running. As such, supplier must provide a function call for Autonertia to invoke  that will refresh failure information on demand. The refreshing function cannot return  control to Autonertia code until the failure information for each PWM output has been  determined. This function call can be combined with the similar function call for digital  outputs. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_SetPWMPolarity ( )

|  |  |
| --- | --- |
| **Description** | Modifier function to set the PWM output polarity |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_** **SetPWMPolarity** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMOut \* | | HWPWMPtr | | | HWPolarityType | | HWPWMPol | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  HW\_SetPWMPolarity() is used to set the polarity of the PWM generated by the  output driver pointed at by HWPWMPtr. If the HWPWMPol is equal to  HW\_LOGIC\_HIGH\_DRIVEN (“OFF” state is low) the duty cycle percentage of the signal  will be driven high. If the HWPWMPol is equal to HW\_LOGIC\_LOW\_DRIVEN (“OFF” state  is high) the duty cycle percentage of the signal will be driven low. Changes to the  polarity of an object should be immediate. | | | | | | **Usage Restrictions:**  Setting of PWM Polarity via this function will be done during initialization phase only.  PWM Polarity will not be changed during run time. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing an instantiated data structure of type HWPWMOut or HWHSPWMOut for each PWM output device, the implementation of the functions as described

above, and any other functions that need to be invoked to support the implementation. The

instantiated data structures will adopt the following naming convention:

HWPWM01 HWHSPWM01

HWPWM02 HWHSPWM02

. .

. .

HWPWMxx HWHSPWMxx

The instantiated structures should be implemented as const data whenever possible. To

properly locate this data, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

If the structures (or any subset of them) need to be implemented in RAM, or for any other RAM needed to fulfill the API, the following pragma statement should be used when declaring the RAM:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that this RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear this RAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required indicating which HWPWMOut/HWHSPWMOut instance is to be used for which actual output. Documentation explaining the diagnostic methodology for each output and any physical output driver shall also be provided. Timing diagrams illustrating any intrusive diagnostic methodologies must be provided.

# PERIOD AND PULSE WIDTH MEASUREMENTS

The Period and Pulse Width Measurement API will consist of a generic interface common to all implementations of PWM input signals. A common data structure instantiated once for each input will be used to manage the interface.

Autonertia will use these two interfaces to measure the duty cycle of the signals. The following

sequence shall be run to determine a signal’s duty cycle:

Initialize the driver for period measurement.

Wait for the period to be completed and store the period.

Initialize the driver for pulse width measurement.

Wait for the pulse to be completed and store the pulse width.

Repeat.

In order to diagnose lost signals, accessor functions are specified to provide the pin state of the signal. Autonertia is responsible for assuring that a period/pulse width has completed before asking for a period/pulse width measurement and that supplier's responsibility to assure coherency is based upon this constraint.

## INTERFACE STRUCTURE

Each PWM input should have an instantiated structure type defined as HWPWMIn. A separate instance of this structure will be created for each input signal. This data structure will be used in the implementation to manage each PWM input signal.

Each signal can be configured for input period measurement or pulse width measurement.

Autonertia is responsible for assuring that signals are configured appropriately before calling

accessor functions to retrieve data. Supplier is responsible for assuring the readings of the

hardware provide coherent results based upon the constraint the Autonertia waits for the

period/pulse to complete.

The following type definition is used to define the polarity of period and pulse width detection

typedef enum

{

HW\_PWMIN\_USE\_FALLING\_EDGE,

HW\_PWMIN\_USE\_RISING\_EDGE

} HWPWMInputEdgeSelectType;

## HW\_InitPeriodMeasurement ()

|  |  |
| --- | --- |
| **Description** | Configures the signal for measuring input period |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitPeriodMeasurement** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | | HWPWMInputEdgeSelectType | | PWMInputEdgeConfig | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | | enum | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will configure the device pointed at by PWMInput for input period  measurement. Input period measurements will measure the period as defined by  PWMInputEdgeConfig. If PWMInputEdgeConfig == HW\_PWMIN\_USE\_RISING\_EDGE  the period will be measured from rising to rising edge. If PWMInputEdgeConfig ==  HW\_PWMIN\_USE\_FALLING\_EDGE the period will be measured from falling to falling edge. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_InitPWMeasurement ()

|  |  |
| --- | --- |
| **Description** | Configures the signal for measuring pulse width |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitPWMeasurement** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | | HWPWMInputEdgeSelectType | | PWMInputEdgeConfig | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | | enum | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will configure the device pointed at by PWMInput for input pulse width  measurement. Pulse width measurements will reflect the last pulse of the duty cycle as  defined by PWMInputEdgeConfig. If PWMInputEdgeConfig ==  HW\_PWMIN\_USE\_RISING\_EDGE the pulse will be measured from rising to falling edge. If  PWMInputEdgeConfig == HW\_PWMIN\_USE\_FALLING\_EDGE the pulse will be  measured from falling to rising edge. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_InitDCMeasurement ()

|  |  |
| --- | --- |
| **Description** | Configures the signal for measuring input duty cycle |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitDCMeasurement** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | | HWPWMInputEdgeSelectType | | PWMInputEdgeConfig | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | | enum | n/a | n/a | n/a | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function will configure the device pointed at by PWMInput for input duty cycle  measurement. Input duty cycle measurements will measure the duty cycle as defined by  PWMInputEdgeConfig. If PWMInputEdgeConfig == HW\_PWMIN\_USE\_RISING\_EDGE  then duty cycle will be measured by ratio of high time to total time. If  PWMInputEdgeConfig == HW\_PWMIN\_USE\_FALLING\_EDGE then duty cycle will be  measured by ratio of low time to total time. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_GetPWMPeriod ()

|  |  |
| --- | --- |
| **Description** | Returns the most recently measured period |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_GetPWMPeriod** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | usec | H=E\*4 | 0 | 4000000 | | **Semantics:**  This function will return the last measured period from the device pointed at by  PWMInput. Input period measurements will measure the period as specified in  HW\_InitPeriodMeasurement(). Supplier is responsible for assuring the readings  of the hardware provide coherent results based on the constraint that Autonertia will wait  for the pulse to complete before requesting a measurement. In the event that a complete  period has never been completed at the time of the function call, an indeterminate value  will be returned. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that period measurements are only requested from  devices configured for period measurement using the function call  HW\_InitPeriodMeasurement() and that the period will not be valid  until the first period has completed after calling this function.  It is understood that, at this high degree of resolution, some deviation will exist between  the requested resolution and the HW capability due to clock frequencies, dividers etc.  Supplier is required to merely re-scale to the output resolution requested. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_GetPulseWidth ()

|  |  |
| --- | --- |
| **Description** | Returns the most recently measured pulse width |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_GetPulseWidth** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | usec | H=E\*4 | 0 | 4000000 | | **Semantics:**  This function will return the last measured pulse width from the device pointed at by  PWMInput. Pulse width measurements will reflect the last pulse of the duty cycle as  specified in HW\_InitPWMeasurement(). Supplier is responsible for assuring the  readings of the hardware provide coherent results based on the constraint that Autonertia  will wait for the pulse to complete before requesting a measurement. In the event that a  complete pulse width has never been completed at the time of the function call, a zero  should be returned. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that pulse width measurements are only requested  from devices configured for pulse width measurement using the function call  HW\_InitPWMeasurement() and that the pulse has completed before calling this  function. It is understood that, at this high degree of resolution, some deviation will exist  between the requested resolution and the HW capability due to clock frequencies,  dividers etc. Supplier is required to merely re-scale to the output resolution requested. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_GetPWMDutyCycle ()

|  |  |
| --- | --- |
| **Description** | Function to get duty cycle of PWM signal |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_GetPWMDutyCycle** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | A pointer of structure as defined below:  typedef struct  {  float DutyCycle  unsigned char DutyStatus,  } tPWM\_Duty; | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | % | n/a | 0 | 100 | | n/a | 1 | 0=Success | 1=Failed | | **Semantics:**  This interface returns the Duty cycle of PWM signal as specified by the channel pointer.  When this function is called with parameter HW\_PWMIN \_RISING\_EDGE, then duty  cycle will be calculated as high-time divided by total-time, but if called with parameter  HW\_PWMIN \_FALLING\_EDGE then duty cycle will be calculated as low-time divided by  total-time. Supplier is responsible for assuring the readings of the hardware provide  coherent results based on the constraint that Chrysler will wait for the duty cycle to  complete before requesting a measurement. In the event that a complete duty cycle has  never been detected at the time of the function call, a zero should be returned. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that pulse width measurements are only requested  from devices configured for pulse width measurement using the function call  HW\_InitPWMeasurement() and that the pulse has completed before calling this  function. It is understood that, at this high degree of resolution, some deviation will exist  between the requested resolution and the HW capability due to clock frequencies,  dividers etc. Supplier is required to merely re-scale to the output resolution requested. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PeriodHasCompleted ()

|  |  |
| --- | --- |
| **Description** | Indicates whether a period has completed |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PeriodHasCompleted** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if a period has completed on the  input pin. A non-zero value will indicate that the period has completed. A zero will be  returned if the period has not completed. The function should clear the status flag  before returning. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that period measurements are only requested from  devices configured for period measurement using the function call  HW\_InitPWMeasurement(). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PulseHasCompleted ()

|  |  |
| --- | --- |
| **Description** | Indicates whether a pulse has completed |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PulseHasCompleted** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if a period has completed on the  input pin. A non-zero value will indicate that the period has completed. A zero will be  returned if the period has not completed. The function should clear the status flag  before returning. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that period measurements are only requested from  devices configured for period measurement using the function call HW\_InitPWMeasurement(). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_DCHasCompleted ()

|  |  |
| --- | --- |
| **Description** | Indicates whether a duty cycle has completed |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_DCHasCompleted** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if a period has completed on the  input pin. A non-zero value will indicate that the period has completed. A zero will be  returned if the period has not completed. The function should clear the status flag  before returning. | | | | | | **Usage Restrictions:**  Autonertia is responsible for assuring that duty cycle measurements are only requested  from devices configured for duty cycle measurement using the function call  HW\_InitDCMeasurement(). | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PWMInputPinIsHigh ()

|  |  |
| --- | --- |
| **Description** | Indicates whether the input pin is in a high state |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PWMInputPinIsHigh** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if the input pin is in a high state. A  zero should be returned if the input pin is in a low state. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PWMInputPinIsLow ()

|  |  |
| --- | --- |
| **Description** | Indicates whether the input pin is in a low state. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PWMInputPinIsLow** | | | | | **Arguments** | **Type** | | **Name** | | | HWPWMIn \* | | PWMInput | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value that will indicate if the input pin is in a high state. A  zero should be returned if the input pin is in a low state. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing an instantiated data structure of type HWPWMIn for each PWM input signal, the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation. The instantiated data structures will adopt the following naming convention:

HWIPWM01

HWIPWM02

HWIPWM03

The instantiated structures should be implemented as const data whenever possible. To

properly locate this data, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

If the structures (or any subset of them) need to be implemented in RAM, or for any other RAM needed to fulfill the API, the following pragma statement should be used when declaring the RAM:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that this RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear this RAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required indicating which HWPWMIn instance is to be used for which actual input. Additionally, documentation explaining the HW resolution capability and expected error for functions that return a time value is required.

# DIRECT MEMORY ACCESS (DMA)

This chapter of the API is provided as a starting point for discussions regarding the shared use of the DMA controller in the microcontroller.

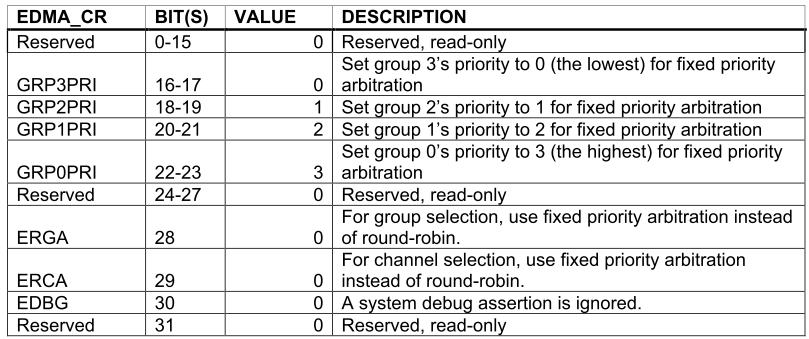
The DMA controller will be used by both Autonertia and supplier software, and its use must be coordinated. The various uses of the DMA controller in Autonertia software with the

corresponding priorities are shown below…

## DMA GLOBAL REGISTERS

A subset of the DMA registers are considered global registers and should be initialized once every power-up with parameters that satisfy DMA usage in both Autonertia and supplier software. Below settings are provided here as a starting point for discussions.

### DMA Control Register (Example)



## DMA Enable Error Interrupt Register (EDMA\_EEIRH, EDMA\_EEIRL)

Each bit of this register corresponds to the error interrupt enable for a particular channel. This register will be initialized to zero; this will disable error interrupts for all DMA channels.

## DMA Enable Request Register (EDMA\_ERQRH, EDMA\_ERQRL)

Each bit of this register implements an enable flag for a particular DMA channel. DMA requests for every channel are disabled during DMA initialization.

## DMA Interrupt Request Register (EDMA\_IRQRH, EDMA\_IRQRL)

Each bit of this register implements an interrupt request flag for a particular DMA channel. All interrupt flags are cleared during initialization.

## DMA Error Register (EDMA\_ERH, EDMA\_ERL)

Each of the DMA error flags are cleared during initialization.

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation.

HWIPWM01

HWIPWM02

HWIPWM03

For any const data required in the implementation, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

For any RAM needed to fulfill the API, the following pragma statements should be used when declaring the RAM, depending upon the storage class required:

#pragma section SDATA ".APIvram"

#pragma section SDATA ".APIdmaram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that VRAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear VRAM to zero. For the code elements, the following pragma statement should be used:

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Supplier shall provide documentation indicating which DMA channels are used and for what

purpose. Additionally, supplier shall provide any time-out limits used in waiting for DMA

transfers to complete, where applicable.

# CONTROLLER SPECIFIC HW DIAGNOSTICS

Supplier is required to provide diagnostics for any hardware or software not specifically called out elsewhere in the API. This portion of the API provides a means of allowing supplier to communicate results of these diagnostic tests to Autonertia software. Examples of diagnostics that are to be interfaced to Autonertia in this manner may include, but are not limited to:

* NVRAM store/restore failures
* DMA errors
* EEPROM Memory Diagnostics

The supplier is provided with 4 following functions, each of which returns a 32-bit value consisting of flags. Each used flag should represent a failure that supplier software has identified that Autonertia will need to take action on. The actions that might ensure from these identified failures include posting faults and/or resetting the controller. Supplier will need to provide the definition of these returned flags, a detailed explanation of what the flag signifies, and an explanation of the diagnostic methodology used in managing the flags. Autonertia software will call these functions at a particular rate and take whatever action is appropriate. Supplier’s interface functions should clear these flags after returning the information.

## HW\_ControllerDiagnostic1 ()

|  |  |
| --- | --- |
| **Description** | Function for obtaining diagnostic information about the controller or supplier’s software. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_ControllerDiagnostic1** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value consisting of flags identifying failures specific to  supplier’s hardware or software not specifically called out elsewhere. Supplier is to  provide the definition of the flags returned by this function. A bit set indicates a  confirmed failure. A bit clear indicates no failure indicated. Unused bits should always  be clear. This function should clear the flags after returning the data so that the results  indicate the failure state since the last time this function was called. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_ControllerDiagnostic2 ()

|  |  |
| --- | --- |
| **Description** | Function for obtaining diagnostic information about the controller or supplier’s software. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_ControllerDiagnostic2** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value consisting of flags identifying failures specific to  supplier’s hardware or software not specifically called out elsewhere. Supplier is to  provide the definition of the flags returned by this function. A bit set indicates a  confirmed failure. A bit clear indicates no failure indicated. Unused bits should always  be clear. This function should clear the flags after returning the data so that the results  indicate the failure state since the last time this function was called. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_ControllerDiagnostic3()

|  |  |
| --- | --- |
| **Description** | Function for obtaining diagnostic information about the controller or supplier’s software. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_ControllerDiagnostic3** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value consisting of flags identifying failures specific to  supplier’s hardware or software not specifically called out elsewhere. Supplier is to  provide the definition of the flags returned by this function. A bit set indicates a  confirmed failure. A bit clear indicates no failure indicated. Unused bits should always  be clear. This function should clear the flags after returning the data so that the results  indicate the failure state since the last time this function was called. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_ControllerDiagnostic4()

|  |  |
| --- | --- |
| **Description** | Function for obtaining diagnostic information about the controller or supplier’s software. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_ControllerDiagnostic4** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | Flag | n/a | n/a | n/a | | **Semantics:**  This function will return a 32-bit value consisting of flags identifying failures specific to  supplier’s hardware or software not specifically called out elsewhere. Supplier is to  provide the definition of the flags returned by this function. A bit set indicates a  confirmed failure. A bit clear indicates no failure indicated. Unused bits should always  be clear. This function should clear the flags after returning the data so that the results  indicate the failure state since the last time this function was called. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## Controller Cause of Reset

The following data structures shall be used in providing controller cause of reset information in case of controller experiencing resets. This section describes the reset type, reset counter and address location of reset.

typedef struct

{

UINT16 HW\_Reset\_PORSCnt; // Power-on reset status.

UINT16 HW\_Reset\_ERSCnt; // External reset status.

UINT16 HW\_Reset\_LLRSCnt; // Loss-of-lock reset status.

UINT16 HW\_Reset\_LCRSCnt; // Loss-of-clock reset status.

UINT16 HW\_Reset\_WDRS0Cnt; // Core 0 Watchdog timer/debug reset status.

UINT16 HW\_Reset\_SWTRS0Cnt; // Software watchdog 0 (SWT0) reset status.

UINT16 HW\_Reset\_SWTRS1Cnt; // Software watchdog 1 (SWT1) reset status.

UINT16 HW\_Reset\_WDRS1Cnt; // Core 1 Watchdog timer/debug reset status.

UINT16 HW\_Reset\_CPURSCnt; // Simultaneous Core0 and Core1 reset status.

UINT16 HW\_Reset\_STCURSCnt; // Self Test control unit reset status.

UINT16 HW\_Reset\_SSRSCnt; // Software system reset status.

UINT16 HW\_Reset\_SERFCnt; // Software external reset flag.

UINT16 HW\_Reset\_RGFCnt; // Reset glitch flag.

} HW\_ResetCauseCount;

typedef enum

{

HW\_Reset\_INIT, // Initialization

HW\_Reset\_PwrOnReset, // Power-on reset status.

HW\_Reset\_ExtReset, // External reset status.

HW\_Reset\_LossOfLockReset, // Loss-of-lock reset status.

HW\_Reset\_LossOfClkReset, // Loss-of-clock reset status.

HW\_Reset\_Core0WDOGReset, // Core 0 Watchdog timer/debug reset status.

HW\_Reset\_Sw\_WDOG0Reset, // Software watchdog 0 (SWT0) reset status.

HW\_Reset\_Sw\_WDOG1Reset, // Software watchdog 1 (SWT1) reset status.

HW\_Reset\_Core1WDOGReset, // Core 1 Watchdog timer/debug reset status.

HW\_Reset\_SimlBothCoreReset, // Simultaneous Core0 and Core1 reset status.

HW\_Reset\_SelfTestReset, // Self Test control unit reset status.

HW\_Reset\_SwSysReset, // Software system reset status.

HW\_Reset\_SwExtReset, // Software external reset flag.

HW\_Reset\_ResetPinGlitchReset // Reset glitch flag.

}HW\_ResetCauseType;

typedef struct

{

HW\_ResetCauseType HW\_CauseofReset; // refer enum definition

HW\_ResetCauseCount HW\_CauseOfReset\_Count; // refer structure

UINT32 HW\_Reset\_StackPtr; // stack pointer location

UINT32 HW\_Reset\_Address; // program or cpu executing

// current instruction

// location causing reset

}HW\_CauseOfReset;

## HW\_InitCauseOfReset()

|  |  |
| --- | --- |
| **Description** | Function for initializing cause of reset parameters. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitCauseOfReset** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function shall initialize the data structure based on user request or upon NVM clear  request for the application. | | | | | | **Usage Restrictions:**  This function shall be called on NVM clear request. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_CauseOfReset()

|  |  |
| --- | --- |
| **Description** | Provides controller cause of reset information. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_InitCauseOfReset** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function shall provide cause of reset information. | | | | | | **Usage Restrictions:**  This function shall be called at the start of controller init (\_main()) to read cause of reset  information. The data structure return value shall be stored in NVM parameter of  structure type HW\_CauseOfReset\*. The function call and return value shall be  maintained by Autonertia software.  Note: During regular Reset-to-KeyOn, HW\_CauseofReset shall always be  HW\_Reset\_INIT | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation.

For any const data required in the implementation, supplier should use the pragma statement:

#pragma section SCONST ".APIconst"

For any RAM needed to fulfill the API, the following pragma statements should be used when declaring the RAM, depending upon the storage class required:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that VRAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear VRAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Supplier shall provide documentation indicating which DMA channels are used and for what

purpose. Additionally, supplier shall provide any time-out limits used in waiting for DMA

transfers to complete, where applicable.

# TASK MANGEMENT – INITIALIZATION, RUNNING, AND POWERDOWN

Autonertia will implement the module initialization, operating system, and powerdown software.

Supplier function calls will be interleaved into Autonertia software where appropriate.

## Clock Frequency

Supplier software shall be responsible for setting system clock frequency. The system clock

setup shall be using FMPLL. The overall clock frequency of the system shall be around ~ (TBD, Agreed upon with supplier) MHz.

## HW\_GetSysClockFrequency ()

|  |  |
| --- | --- |
| **Description** | Allows Autonertia software to read system clock frequency from supplier sw. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_SetSysClockFrequency** | | | | | **Arguments** | **Type** | | **Name** | | | unsigned short int | | Freq | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | MHz | H=E | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  This function shall provide cause of reset information. | | | | | | **Usage Restrictions:**  This function will be called to allow Autonertia software reading system clock frequency  from Supplier’s software. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## Module Initialization

Module initialization entails everything necessary before starting the temporal operating system tasks. The following is a “high level” description of the tasks performed during module initialization with responsibilities assigned between Autonertia and Supplier.

|  |  |  |  |
| --- | --- | --- | --- |
| **Initialization / Configuration** | **Responsibility** |  | **Comments** |
| **Autonertia** | **Supplier** |
| PMC |  | x |  |
| BTB | x |  |  |
| Config Watchdog | x |  |  |
| Config Base Timers | x |  |  |
| Config Interrupt | x |  |  |
| FMPLL Unit | x |  |  |
| MMU Init | x |  |  |
| VRAM ECC Init | x |  |  |
| Flash Memory Array Init | x |  |  |
| XBAR Init | x |  |  |
| SIU Init | x | x |  |
| DMA Init | x | x |  |
| Powerup Init |  | x |  |
| RAM Test | x |  |  |
| Enable Cache & Cache Stack | x |  |  |
| Digital I/O Init |  | x |  |
| EEPROM Mimic RAM Init |  | x |  |
| NVRAM Init | x |  |  |
| Enable Cache for RAM | x | x |  |
| Config Calibration Tool | x |  |  |
| LIN Init | x |  |  |
| ADC Init |  | x |  |
| PWM Init |  | x |  |
| Start RTOS Core 0 | x | x |  |

It is understood that there are numerous Supplier function calls that need to be invoked by

Autonertia software during the module initialization. It’s the Supplier’s responsibility to specify and document the names and sequence of all function calls.

## Temporal Tasks

The Autonertia operating system will provide temporal tasks that run at the following intervals:

Core 0 (Valve/Actuator):

1 ms

2 ms

5 ms

10 ms

25 ms

100 ms

1000 ms

There are two task categories: Urgent and Normal Tasks. The Urgent task is restricted to real-time system management functionality only. The urgent task can only be preempted by either interrupt, kernel or another urgent task with higher priority. Normal tasks have a defined priority scheme such that the task can be selected and arbitrated based on its relative importance to the system. Low priority tasks may be shed (ignored) during heavy CPU loads, to help maintain a consistent system time base during high CPU burden. They will be automatically recovered as CPU load decreases. Normal task executes as run-to-completion, and can be preempted by all interrupts, urgent tasks and kernel.

Supplier’s function calls could be invoked by Autonertia Operating System. Supplier needs to

provide following information in order to make function calls in any of these tasks:

* Execution rate (schedule interval)
* Task category (Urgent or Normal)
* Task priority (High or Low, normal task only)

Supplier is encouraged to use normal task and the slowest possible task rate whenever possible to preserve bandwidth.

## Powerdown

The Autonertia operating system enters the powerdown state when the ignition run/start switch is turned off by the driver. During this time, I/O is shutdown in preparation for module powerdown. However, due to the numerous tests that are performed during this time, three temporal tasks continue to run. These are the1, 2 and 5 ms tasks. Supplier software for EEPROM writing, digital inputs, digital output, and PWM outputs will continue to be called during this time for select signals. It’s Supplier software’s responsibility to configure and enable the system wakeup before system is powerdown.

Note that if the driver turns the key back on during this time, a final “clean-up” is performed and the module is physically reset to begin the next drive cycle.

It is understood that there are numerous function calls that supplier will need to have called

during the powerdown state.

## Miscellaneous Functions

## HW\_PowerUpInit ()

|  |  |
| --- | --- |
| **Description** | Perform Initialization for PAD configuration and Turn on the power Supply |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PowerUpInit** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | supplier defined | n/a | n/a | | **Semantics:**  This function will be called to initialize the PADS configuration registers. Note that  supplier is expected to also assure that output pins are configured to the correct state  before configuring the pins to outputs.  This function will also configure the power supply to remain active until commanded off  even if the ignition key is turned off. This function should not return control to Autonertia  software until verification of the hold is made or failure of the hold is determined. The  return value should indicate success or the details of the failure (i.e., SPI communication  timeout, SPI communication message failure, power-supply failure, etc.). This function  will be called before supplier is called to initialize most hardware so the function should  be able to be called independently. | | | | | | **Usage Restrictions:**  None. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_WasBatteryDisconnected ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding whether the module power was lost since the software  last ran. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_WasBatteryDisconnected** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | 0=False  Not 0 = True | n/a | n/a | | **Semantics:**  This function will be called to identify if power has been removed from the module since  the prior key-on event. A non-zero returned value will indicate that the battery was  disconnected. A zero will be returned if battery was not disconnected. This function  should clear any indication in the hardware that the battery was disconnected. | | | | | | **Usage Restrictions:**  Device which supplies the information must be initialized. | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PSInitializationStatus ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding the initialization state of the power supply. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PSInitializationStatus** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | Supplier defined | n/a | n/a | | **Semantics:**  This function will be called to at initialization to ascertain the initialization state of the  power supply. Information from the power supply should include, but not necessarily be  limited to, information on the cause of reset (if applicable) and wakeup source. This  function will be called before supplier is called to initialize most hardware so the function  should be able to be called independently. Note that any RAM used by supplier during  this call will be cleared to zero when RAM integrity tests are complete. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_PSStatus ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding the initialization state of the power supply. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PSStatus** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | Supplier defined | n/a | n/a | | **Semantics:**  This function will be called to at initialization to ascertain the initialization state of the  power supply. Information from the power supply should include, but not necessarily be  limited to, information on the cause of reset (if applicable) and wakeup source. This  function will be called before supplier is called to initialize most hardware so the function  should be able to be called independently. Note that any RAM used by supplier during  this call will be cleared to zero when RAM integrity tests are complete. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_TurnPowerSupplyOff ()

|  |  |
| --- | --- |
| **Description** | Allows power supply to de-activate if the ignition key is turned off |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_TurnPowerSupplyOff** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | Supplier defined | n/a | n/a | | **Semantics:**  This function will be called to allow the power supply to deactivate if the ignition key is  turned off. This function should not return control to Autonertia software until verification  of the hold release is made or failure of the hold release is determined. The return value  should indicate success or the details of the failure (i.e., SPI communication timeout, SPI  communication message failure, power-supply failure, etc.). | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## HW\_APIVersion ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding supplier’s version of software |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_APIVersion** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char \* | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | | **Semantics:**  This function will be called to obtain version information of supplier’s software. This data  should be stored by supplier into an array of 8 ASCII characters. Supplier must provide  the definition of their API version identifier in the documentation for each software  release. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation.

For any const data required in the API, supplier should use the pragma statement when declaring const data:

#pragma section SCONST ".APIconst"

For any RAM needed to fulfill the API, the following pragma statements should be used when declaring the RAM:

#pragma section SDATA ".APIvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required for all supplier functions that need to be invoked during initialization, during temporal tasks and engine events, and during powerdown. This documentation should clearly state the function name, the expected function run-time, at what point the software should be called, and preliminary sequencing considerations that need to be taken into account.

# BOOTLOADER INTERFACES

In order to implement bootloader software, Autonertia will need access to supplier’s EEPROM

mechanisms, system reset capability and module power control. Bootloader is a separately

released software component so the software elements must be provided independently from those software elements that make up the application software.

Memory management operations performed in the bootloader software must respect the

memory management operations performed in the application software. EEPROM (or EEPROM emulation) must work seamlessly, whether the operations are called from bootloader or application software. One noteable difference between the bootloader and application EEPROM interface is that writes to EEPROM from bootloader software will not run in background over several loops. Function calls performing writes to EEPROM from bootloader software will not return until the write is complete, or has been determined to have failed.

Refer to chapter 4 - Memory Management for details on the application memory management interface and structure. ~~A battery backed NVRAM will strategy will be used.~~

## Interface Structure

The following data structure is employed in managing the EEPROM:

typedef struct

{

unsigned char\* RAMStartAddress;

unsigned short int EEOffset;

unsigned short int TableSize;

unsigned short int CheckSum;

} HWEETableStruct;

where RAMStartAddress is a pointer to the address in mimic RAM corresponding to the

EEOffset in the EEPROM. TableSize is the number of bytes that should be manipulated.

CheckSum is calculated as 2’s complement of sum of high word of RAMStartAddress, low

word of RAMStartAddress, EEOffset, and TableSize. This structure represents a

contiguous block of memory that will either be read out of the EEPROM into RAM or written

from RAM to the EEPROM.

Supplier is encouraged to optimize the underlying implementation of physically storing/reading the data by identifying from the addressing and size information when larger data elements can be written or read to/from physical storage, thereby reducing the total time necessary to complete the transfer. Additionally, in the event that physical storage for some data requested to be written is already equal to the requested data, no physical store needs to be made redundantly.

Supplier before writing to Physical EEPROM will have to verify the data in HWEETableStruct using checksum. Verification is successful if sum of high word of RAMStartAddress, low word of RAMStartAddress, EEOffset, TableSize and checksum is Zero.

## BL\_InitializeEEMimicRAM ()

|  |  |
| --- | --- |
| **Description** | Function to initialize mimic RAM from EEPROM storage during bootloader operation |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_InitializeEEMimicRam** | | | | | **Arguments** | **Type** | | **Name** | | | HWEETableStruct \* | | TablePtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | 0 | 1 | | **Semantics:**  0 = HW\_EEPROM\_INIT\_SUCCESS  1 = HW\_EEPROM\_INIT\_FAILED  This function will initialize the RAM specified by TablePtr.RAMStartAddress and  TablePtr.TableSize from the EEPROM storage specified by  TablePtr.EEOffset. In the event that the transfer of data fails for any reason, the  function will write 0xFF to all bytes specified by the function argument and return the  appropriate value to indicate failure. In the event that an EEPROM emulation strategy is  employed, this operation will need to coordinate strategy with the EEPROM emulation  strategy employed in the application software. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | Return value will indicate success/fail. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_InitiateEEPROMWrite ()

|  |  |
| --- | --- |
| **Description** | Function to request storage of RAM to EEPROM |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_InitiateEEPROMWrite** | | | | | **Arguments** | **Type** | | **Name** | | | HWEETableStruct \* | | TablePtr | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned long | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | supplier defined | n/a | n/a | | **Semantics:**  This function will store the RAM specified by TablePtr.RAMStartAddress and  TablePtr.TableSize into the EEPROM storage specified by  TablePtr.EEOffset. Supplier will provide return value definitions to elaborate  success and all failure conditions including, but not limited to, SPI communications  timeout, input parameter invalid, verification failed, etc. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | Return value will indicate success/fail. Unlike the application EEPROM mechanism, this  function is not to return until write is either successful or it is identified that write has  failed. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_PowerUpInit ()

|  |  |
| --- | --- |
| **Description** | Perform Initialization for PAD configuration and Turn on the power Supply |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **HW\_PowerUpInit** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | supplier defined | n/a | n/a | | **Semantics:**  This function will be called to initialize the PADS configuration registers. Note that  supplier is expected to also assure that output pins are configured to the correct state  before configuring the pins to outputs.  This function will also configure the power supply to remain active until commanded off  even if the ignition key is turned off. This function should not return control to Autonertia  software until verification of the hold is made or failure of the hold is determined. The  return value should indicate success or the details of the failure (i.e., SPI communication  timeout, SPI communication message failure, power-supply failure, etc.). This function  will be called before supplier is called to initialize most hardware so the function should  be able to be called independently. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_TurnPowerSupplyOff ()

|  |  |
| --- | --- |
| **Description** | Allows power supply to de-activate if the ignition key is turned off |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_TurnPowerSupplyOff** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | supplier defined | n/a | n/a | | **Semantics:**  This function will be called to de-activate the power supply if the ignition key is turned off.  This function should not return control to Autonertia software until verification of the hold  release is made or failure of the hold release is determined. The return value should  indicate success or the details of the failure (i.e., SPI communication timeout, SPI  communication message failure, power-supply failure, etc.). | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_WasBatteryDisconnected ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding whether the module power was lost since the software  last ran. |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_WasBatteryDisconnected** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | enum | supplier defined | 0 | 1 | | **Semantics:**  0 = FALSE  Non-zero = TRUE  This function will be called to identify if power has been removed from the module since  the prior key-on event. A non-zero returned value will indicate that the battery was  disconnected. A zero will be returned if battery was not disconnected. This function  should not modify any indication in the hardware that the battery was disconnected as  the application function HW\_WasBatteryDisconnected()will subsequently be  called to ascertain the information as well. | | | | | | **Usage Restrictions:**  Device which supplies the information must be initialized. | | | | | |
| **Error Handling** | None. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_SystemReset ()

|  |  |
| --- | --- |
| **Description** | Function to reset the entire controller |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_SystemReset** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | void | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | | **Semantics:**  Performs whatever actions are necessary to reset the entire controller, including the  main microprocessor, the monitoring microprocessor, and all attached devices that  influence/control I/O. The function should never return and should attempt to reset the  controller repeatedly. It is intended that this function takes as direct a path as possible to  resetting the controller. | | | | | | **Usage Restrictions:**  Invoked only by Autonertia software. | | | | | |
| **Error Handling** | None. |
| **Variablity Allowances** | |  | | --- | | Size: Hardware Dependent | | Speed: As fast as possible | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## BL\_APIVersion ()

|  |  |
| --- | --- |
| **Description** | Provides information regarding supplier’s version of bootloader software |
| **Resources Provided** | **Syntax** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Function Name** | **BL\_APIVersion** | | | | | **Arguments** | **Type** | | **Name** | | | void | |  | | |  | |  | | |  | |  | | | **Argument Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | n/a | n/a | n/a | n/a | |  |  |  |  | |  |  |  |  | | **Return Value Type** | unsigned char \* | | | | | **Return Value Scaling** | **Units** | **Scaling** | **Low Lim** | **High Lim** | | pointer | n/a | n/a | n/a | | **Semantics:**  This function will be called to obtain version information of supplier’s bootloader software.  This data should be stored by supplier into an array of 8 ASCII characters. Supplier must  provide the definition of their API version identifier in the documentation for each  software release. | | | | | | **Usage Restrictions:**  None | | | | | |
| **Error Handling** | None. |
| **Variablity Allowances** | |  | | --- | | Size: | | Speed: | | Other: | |
| **Target:** | MPA, Valve |
| **Usage Guide:** | Implied |

## DELIVERABLES

### OBJECT FILES

Supplier will deliver object files containing the implementation of the functions as described above, and any other functions that need to be invoked to support the implementation. Object files from the supplier for the bootloader application interface are to be delivered in a library file as defined in the chapter TBD.

In the event that supplier utilizes functions from the C Library provided with the compiler,

documentation reflecting this use should be provided detailing in what functions the libraries were used, which libraries were used, and which functions from the libraries were used. In the event that supplier utilizes different third-party libraries, supplier must provide a copy of the library for Autonertia use in addition to those items requested above.

For any const data required in the API, supplier should use the pragma statement when declaring const data:

#pragma section SCONST ".API\_BLconst"

For any RAM needed to fulfill the API, the following pragma statements should be used when declaring the RAM:

#pragma section SDATA ".API\_BLvram"

Supplier is responsible for any software needed to initialize RAM. Autonertia will be responsible for assuring that RAM is cleared to zeroes upon initialization before supplier initialization code is run. Supplier initialization code need not specifically clear RAM to zero.

For the code elements, the following pragma statement should be used:

#pragma section CODE ".APIcode"

### HEADER FILES

Supplier will also provide an associated header file containing external declarations of each of the above structures and any function prototypes needed. The file naming convention for

interface files is defined in the chapter on “Header File Naming Convention”. At the

implementer’s discretion, functions can be in-lined and implemented in the header.

### SOURCE CODE

Supplier will provide source code for all functions developed in order to support Autonertia testing tools.

### DOCUMENTATION

Documentation is required for the following items:

* Physical reset capability and how the software invokes the reset on the hardware.
* Details of how module power is controlled.
* Details of how EEPROM (or EEPROM emulation) is managed.
* Overview of NVRAM storage strategy.

# PROJECT SPECIFIC INFORMATION

## Directory Structures

In order to support debugging of supplier’s software and Autonertia tools, source code for all

specified functions is required. Supplier must supply the source code in a directory structure

that matches the directory structure supplier used to compile the software (from the first root subdirectory out to the last subdirectory). Supplier must also indicate the actual drive letter that the directory structure resided under. This will enable Autonertia to set-up an environment that will allow the tools to work correctly. Autonertia’s preference is that the actual drive compiled under would be the C: drive, but it is recognized that this is not always possible.

## Header File Naming Convention

To simplify this document for multiple suppliers, file names for the interface files for each

chapter are defined here. Note that these should only contain the functional interfaces and

should only contain those function prototypes and enumerations necessary to call the functions defined in the interface. Object instantiations for the various interfaces will be defined in a separate header file as outlined in the next section.

|  |  |
| --- | --- |
| **Chapter** | **Interface File Name** |
| Memory Management | api\_mem.h |
| Interrupts & Exceptions | api\_ints.h |
| Controlled Reset | api\_rset.h |
| Analog to Digital Conversions | api\_a2d.h |
| Digital Inputs | api\_digi.h |
| Digital Outputs | api\_digo.h |
| PWM Outputs | api\_pwmo.h |
| Period and Pulse Width Measurements | api\_ipwm.h |
| SPI Device Interfaces | api\_spi.h |
| Direct Memory Access (DMA) | api\_mem.h |
| Controller Specific HW Diagnostics | api\_diag.h |
| Task Management – Initialization, Running, and Powerdown | api\_task.h |
| Bootloader Interfaces | api\_btld.h |
| Miscellenous Interfaces | api\_msc.h |

Any additional header files that are provided with the interface must be named with the pattern:

api\_xxxxx.h

## Libraries

Separate libraries containing the object files from supplier will be required for each controller supported. The object files and library must be compatible for linking with the WindRiver 5.8 compiler.

For application level software, the file name should be:

api\_bpcm.lib

For bootloader interfaces, the file name should be:

api\_bpcmbl.lib

Autonertia toolchain will use C++ compiler/linker to compile/link all C source files by default. In

order to avoid name mangling, Supplier should use the \_\_cplusplus preprocessor macro to

determine which language is being compiled, and then declare these functions with C linkage if being used from a C++ language module. The example is provided below:

// MyCFuncs.h

#ifdef \_\_cplusplus

extern "C" { // only need to export C interface if

// used by C++ source code

#endif

void MyCFunc();

void AnotherCFunc();

#ifdef \_\_cplusplus

}

#endif

In the event that supplier utilizes functions from the C Library provided with the compiler,

documentation reflecting this use should be provided detailing in what functions the libraries

were used, which libraries were used, and which functions from the libraries were used. In the event that supplier utilizes different third-party libraries, supplier must provide a copy of the library for Autonertia use in addition to those items requested above. Any code that the suppliers use from libraries that is deemed relevant to the Autonertia will need to be allocated into the memory section reserved for function.

## General Comments on Deliverables

With each delivery of the realized API, supplier will provide a detailed description of all

components that are incomplete or missing and the dates at which their completion is expected.