

RZ/T2M Group

RZ/T2M Motor Solution Kit Firmware (Motor Control, EtherCAT)

Introduction

This application note describes the firmware of the RZ/T2M Motor Solution Kit equipped with MPU of RZ/T2M group by Renesas Electronics Corporation.

This application note describes the firmware specification of the motor control part of CPU0 of RZ/T2M and the EtherCAT (CiA402) part of CPU1.

Target Device

RZ/T2M Group

When applying the sample program covered in this application note to another microcomputer, modify the program according to the specifications for the target microcomputer and conduct an extensive evaluation of the modified program.

List of Abbreviations and Acronyms

Abbreviation	Full Form
bps	bits per second
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
Hi-Z	High Impedance
I/O	Input / Output
LSB	Least Significant Bit
MSB	Most Significant Bit
NC	Non-Connect
PWM	Pulse Width Modulation
SFR	Special Function Register
UART	Universal Asynchronous Receiver/Transmitter

Contents

1. Introduction.....	4
1.1 Summary	4
1.2 Function.....	4
1.3 Firmware Configuration	5
2. Operating Environment.....	6
3. File Configuration	7
4. Firmware Architecture.....	10
4.1 Overview.....	10
4.1.1 Startup Functions	12
4.1.2 Non-Real-Time Functions	13
4.1.3 Periodic, Real-Time Functions	14
4.1.4 Communication Functions.....	15
4.2 Data Types	16
4.3 Data Structures and Variables	17
4.4 Enumerations, Macros and Constants	17
5. Initialization and Startup Functions	18
5.1 Bootloader	18
5.2 Peripherals Initialization	18
5.3 Firmware Initialization.....	19
6. Servo Control Operation	20
6.1 Motor Position - Encoder Interface.....	21
6.2 Motor Control - Torque Generator.....	22
6.3 Position Control – PID Regulator	25
6.4 Motion Planning - Velocity Profile Generator (Not supported)	27
6.5 Motion Control Parameters (Not Supported).....	30
6.5.1 Target Position	30
6.5.2 Maximum Velocity	30
6.5.3 Maximum Acceleration and Deceleration.....	30
6.5.4 Maximum Acceleration and Deceleration Jerk.....	31
6.5.5 Motion Start Modes	31
6.5.6 Motion Stop Modes	31
7. System Control Functions.....	32
7.1 Interlocks	32
7.2 Electronic Gearing(Non Supported)	34
7.3 Digital Inputs and Outputs	35

RZ/T2M Group	RZ/T2M Motor Solution Kit Firmware (Motor Control, EtherCAT)
7.4	Data Recording..... 35
7.5	Motor Phasing 38
7.6	Motor Homing 39
8.	Host Communication 41
8.1	ASCII Communication Protocol..... 41
8.2	Binary Packet Communication Protocol..... 42
8.2.1	Communication Bus 42
8.2.2	Modules Addressing 43
8.2.3	Request Packet Format..... 43
8.3	EtherCAT Support 45
8.3.1	Software Configuration..... 45
8.3.2	CiA402 Drive Profile 46
8.3.3	Shared Memory 57
9.	Related Documents 69
Appendix A	ASCII Communication Protocol Commands 70
Appendix B	Packet Communication Protocol Commands..... 76
B-1	Get Module Data (Packet Code: 0) 77
B-2	Status Report Request (Packet Code: 1)..... 78
B-3	Function Call (Packet Code: 2) 80
B-4	Initialize PVT Stream (Packet Code: 3)..... 80
B-5	Stream PVT Data (Packet Code: 4) 81
B-6	Set Parameter (Packet Code: 5) 81
B-6.1	Set 16-bit Parameter 81
B-6.2	Set 32-bit Parameter 82
B-7	Get Parameter (Packet Code: 6)..... 82
B-8	Report Trace Buffers Content (Packet Code: 7) 83
B-9	Set Communication Baud Rate (Packet Code: 15)..... 84
Revision History 85

1. Introduction

1.1 Summary

The RZ/2M Solution Kit Firmware is an embedded application that implements the functions of a full featured dual-channel industrial servo controller. The reference code demonstrates the motion control capabilities of the Renesas RZ/T2M device including its high-performance deterministic CPU core, flexible absolute encoder interface and variety of connectivity options.

1.2 Function

The firmware implements the following main functions:

- Initialization of the RZ/T2M cores and its peripherals:

Executable code is transferred from the QSPI Flash memory to the device RAM. The configurable hardware is initialized to support the preferred absolute encoder interface. The Delta Sigma Interface (DSMIF) and the Timers are configured to interface with the dual channel Inverter. PWM Timer Interrupt handler is setup to invoke the Real-time control functions. The SCI Interrupt handlers are setup to respond to host commands. The GPIO pins are initialized to interact with the different digital inputs and outputs of the RZ/T2M Motor Solution Board (Solution board).

- Processing of the host commands

The firmware supports two communication protocols concurrently – ASCII Command Protocol and Binary Packet Protocol. The command interpreter detects the type of the commands and invokes the appropriate dispatcher. The firmware recognizes over 100 commands providing access to all control parameters and algorithms. The host can obtain information periodically to track the status of each motor and control the execution of motion requests. Alternatively, the host can configure the collection of samples from different variables that can be buffered on the devices and analyzed at a later time.

- Processing of the control loops algorithms in real-time

The control algorithms are invoked from the context of the timer that generates the PWM timing (50us). This ensures deterministic – real-time performance of the control function. The real-time tasks include obtaining the current position, executing the position control loop, executing the current control loop (Field Oriented Control), generating the duty cycle for the next PWM period and finally collecting data for future diagnostics.

1.3 Firmware Configuration

This firmware consists of motor control firmware and two libraries. The ECL library is utilized to initialize and interface with different encoder protocols. The Velocity Profile Generation (VPG) library is used to generate a series of set points needed by facilitate point-to-point motion.

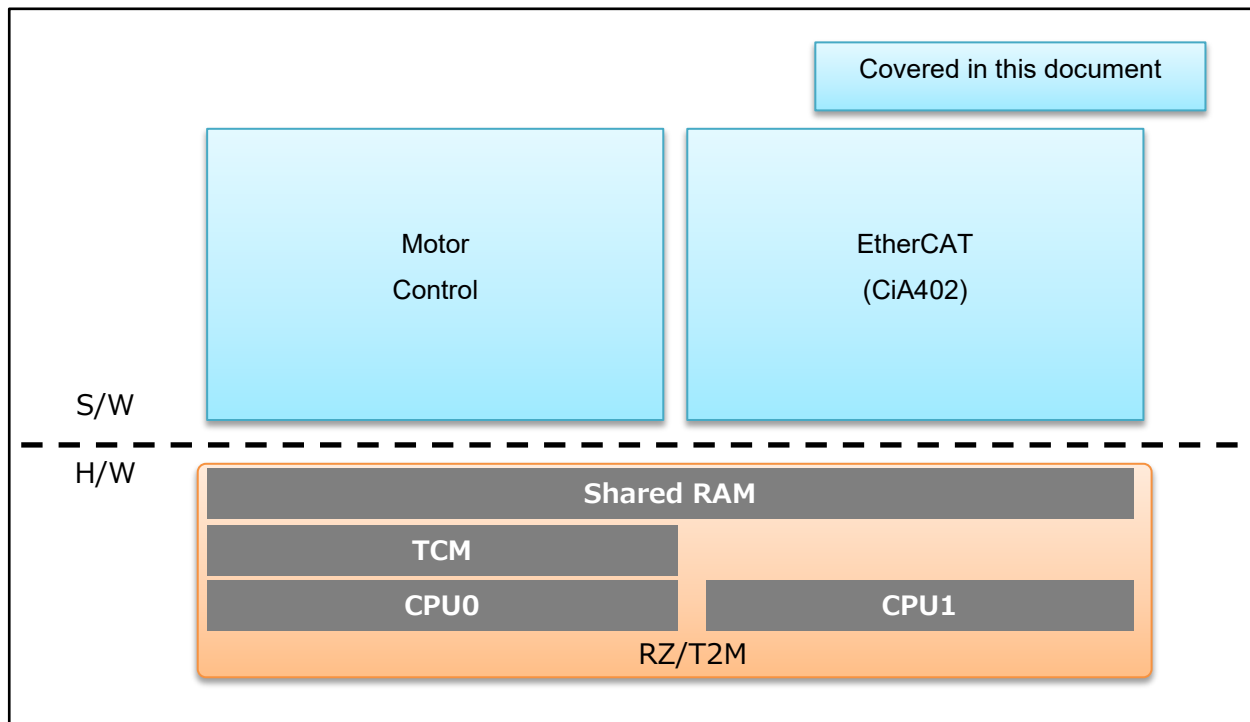


Figure 1.1 RZ/T2M Motor Solution Kit Software Configuration Diagram

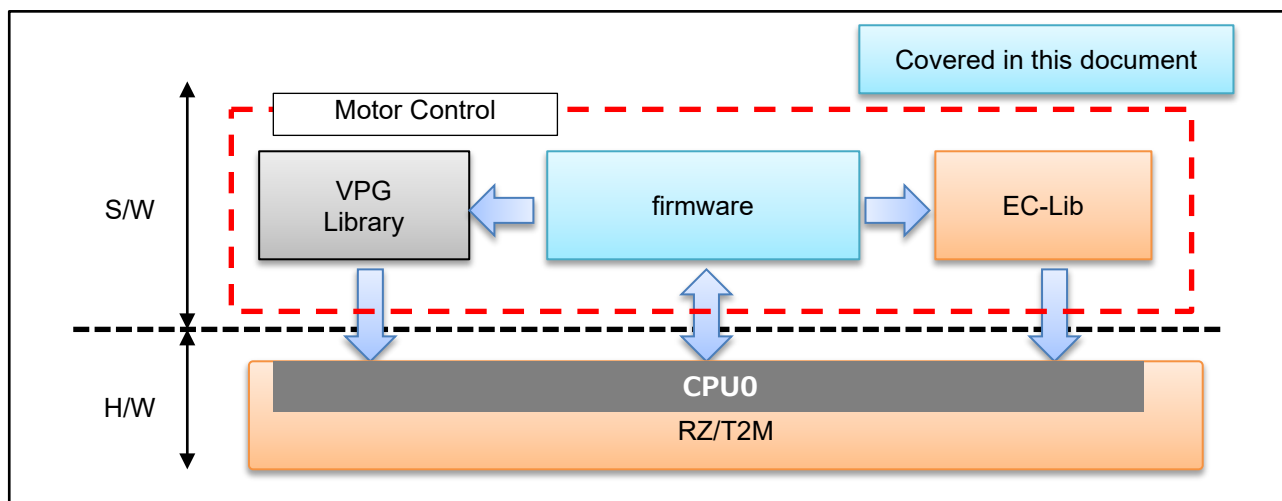


Figure 1.2 RZ/T2M Motor Solution Kit Software Configuration Diagram (Motor Control detail)

2. Operating Environment

Operating Environment refer the following documents.

- RZ/T2M Motor Solution Kit Startup Manual(for RZ/T2M Motion Control Utility) (R01AN5989)
- RZ/T2M Motor Solution Kit Startup Manual (for EtherCAT) (R01AN6470)

3. File Configuration

The figure below shows the file configuration of the Solution Kit Firmware:

Table 3.1 File Configuration (CPU0)

File	Description
inc\apl\m_common.h	The Solution Kit Firmware header file. Includes the motor data structure and signatures of all global functions.
inc\apl\m_commands.h	Include file of m_commands.c
inc\apl\m_interpreter.h	Include file of m_interpreter.c
inc\apl\m_recorder.h	Include file of m_recorder.c
inc\common\r_typedefs.h	Common types definitions header file
inc\common\platform.h	Common types definitions header file
inc\scifa_uart	The files in this directory are header files for serial communication
inc\serial_flash	The files in this directory are header files for QSPI flash access
inc\shm	The files in this directory are header files for shared memory access
src\drv\m_biplane.h	Common types definitions header file
src\drv\m_rzt.c	Code specific for the hardware on the Solution Kit and the RZ/T2M device
src\drv\m_inputs.c	Code specific for the digital input functions on the Solution Kit
src\drv\scifa_uart	The files in this directory are code specific for serial communication
src\drv\serial_flash	The files in this directory are code specific for QSPI flash access
src\drv\shm\r_shm.c	Code specific for shared memory access
src\drv\dsdif	The files in this directory are header files for delta sigma interface
src\apl\m_commands.c	The code for all host commands that can be invoked
src\apl\m_commutation.c	The code for the motor commutation algorithms such as Space Vector Modulation, Hall-based Trapezoidal Commutation
src\apl\m_control.c	The real-time algorithms control execution branches dependent on different state and configuration options
src\apl\m_homing.c	The state machine implementing the homing algorithm
src\apl\m_interlocks.c	The functions checking various interlock conditions.
src\apl\m_interpreter.c	The command parser for the ASCII commands and the command decoder for the binary packets.
src\apl\m_phasing.c	The functions implementing the different phasing algorithms.
src\apl\m_pid_calc.c	The Position control loop algorithm implementation
src\apl\m_pos_read.c	The encoder position reading control algorithm
src\apl\m_recorder.c	The data collection functions and the start / stop triggers evaluation
src\apl\m_vpg_trap.c	The Trapezoidal Velocity Profile Generator
src\encoder\BiSS	The files in this directory are dedicated to the implementation of the interface to the BiSS-C absolute encoder interface protocol
src\encoder\EnDat	The files in this directory are dedicated to the implementation of the interface to the EnDat 2.2 absolute encoder interface protocol
src\encoder\AFormat	The files in this directory are dedicated to the implementation of the interface to the A-format™ absolute encoder interface protocol

src\encoder\FACoder	The files in this directory are dedicated to the implementation of the interface to the FA-Coder absolute encoder interface protocol
src\encoder\HIPERFACE_DSL	The files in this directory are dedicated to the implementation of the interface to the HIPERFACE DSL absolute encoder interface protocol
src\encoder\r_enc_int_rzt2.h	Macros and function definitions of encoder interface library
src\encoder\r_enc_int_rzt2.c	Encoder I/F common functions
src\sharedmemory\shm_access.c	Code specific for shared memory access for motor control via EtherCAT communication
src\sharedmemory\shm_motor.c	Code specific for motor control via EtherCAT communication
dat\Config_AFormat_v1.0.dat	Configuration file for the A-format absolute encoder communication protocol.
dat\Config_Biss_v1.0.dat	Configuration file for the BiSS absolute encoder communication protocol.
dat\Config_Endat22_v1.0.dat	Configuration file for the EnDat absolute encoder communication protocol.
dat\Config_Fa_Coder_V1.0.dat	Configuration file for the FA-Coder absolute encoder communication protocol.
dat\Config_hfdsl_V1.0.dat	Configuration file for the HIPERFACE DSL absolute encoder communication protocol.
lib\libVPG.a	The Velocity Profile Generation Library for gcc
lib\ecl\r_ecl_rzt2_gcc.a	The Configurable hardware initialization library for gcc. It is used to facilitate the loading of configuration that matches the specifics of the selected absolute encoder interface protocol
lib\ecl\r_ecl_rzt2_if.h	Encoder I/F library header file for gcc
lib\ecl\RZT2_pinmux_v1.0.bin	PINMUX configuration data for gcc
lib\iar\r_vpg.a	The Velocity Profile Generation Library for IAR
lib\iar\r_ecl_rzt2_iar.a	The Configurable hardware initialization library for IAR. It is used to facilitate the loading of configuration that matches the specifics of the selected absolute encoder interface protocol
lib\iar\r_ecl_rzt2_if.h	Encoder I/F library header file for IAR
lib\iar\RZT2_pinmux_v1.0.bin	PINMUX configuration data for IAR
cg_src	They initialize the different RZ/T2M peripherals involved in the firmware operation.
src	main function
CPU1_boot_bin\RZT2_esc_cpu1.bin	Binary file to be output from CPU1 project

Table 3.2 File Configuration (CPU1)

File	Description
application\ecat	The files in this directory are Slave Stack Codes generated by the SSC Tool
rzt\arm\CMSIS_5\CMSIS\Core_R\Include	The files in this directory are CMSIS Cortex-R52 header files
rzt\board\	The files in this directory for board
rzt\fsp\inc	Include files of generated drivers by FSP
rzt\fsp\src\bsp	BSP files
rzt\fsp\src\cmtw	CMTW driver
rzt\fsp\src\ecat\hal	ESC (EtherCAT Slave Controller) HAL driver
rzt\fsp\src\ecat\phy	PHY driver
rzt\fsp\src\ecat\utilities\batch_files\apply_patch.bat	The batch file to apply a patch file on the Slave Stack Code
rzt\fsp\src\ecat\utilities\batch_files\RZT2_CiA402.patch	The patch file to be applied on the Slave Stack Code
rzt\fsp\src\ecat\utilities\esi\Renesas_RZT2M_Motor_Solution_Kit_CiA402.xml	EtherCAT Slave Information file
rzt\fsp\src\ecat\utilities\ssc_config\Renesas_RZT2_config.xml	SSC Tool EtherCAT Slave Project file
rzt\fsp\src\ioport	I/O port driver
src\drv	Header files and code for shared memory access
src	CPU1 main function

4. Firmware Architecture

4.1 Overview

This Firmware architecture is designed as a set of modules with specific purpose, operating on data structures passed as arguments. The use of global variables is avoided whenever possible. The functions are invoked in a strictly defined priority and context. This is intended to guarantee the deterministic and robust operation of the control algorithms.

The firmware operations are partitioned in different domains based on their need for deterministic behavior. The figure bellow illustrates these domains:

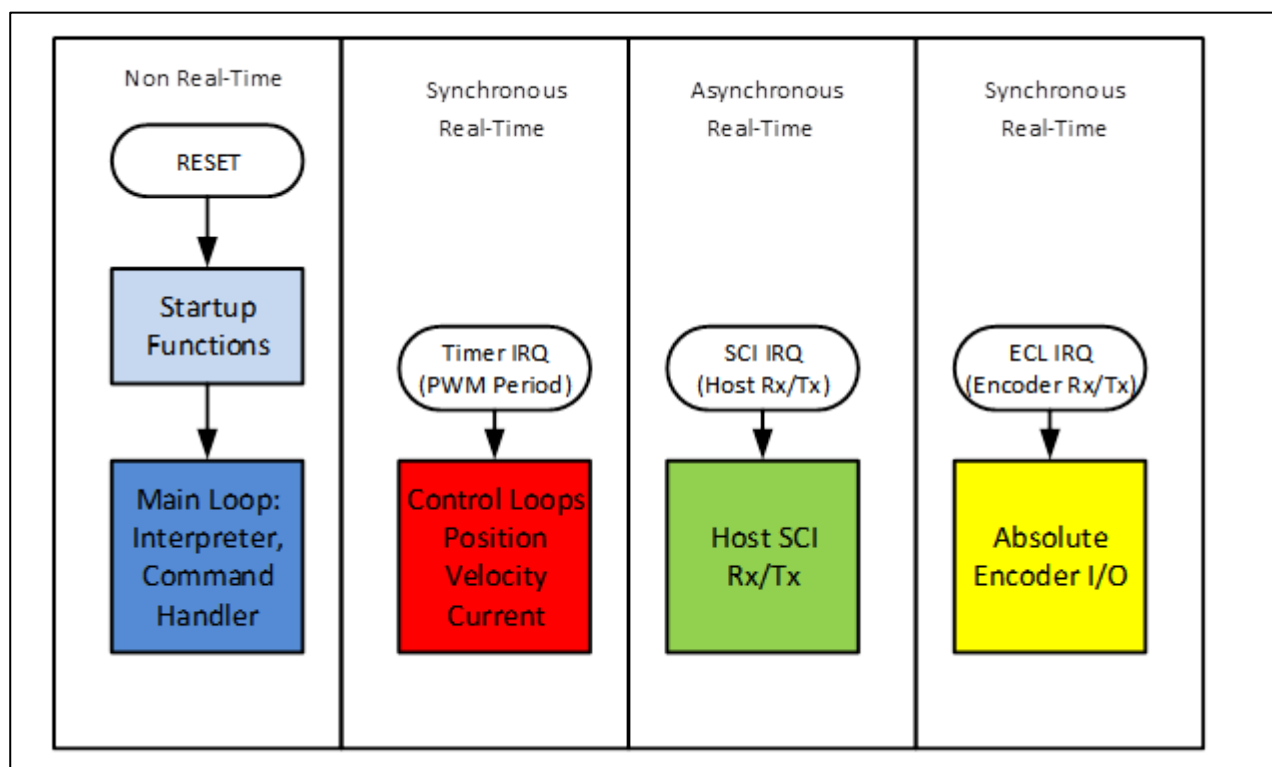


Figure 4.1 Firmware code execution domains.

The operation of the above functions at run-time can be presented in a time diagram that shows their execution flow. The communication tasks overlap with the others because the RZ/T2M employs a dedicated hardware block to interface with the absolute encoders and a FIFO buffer to serve the SCI communications with minimum CPU participation.

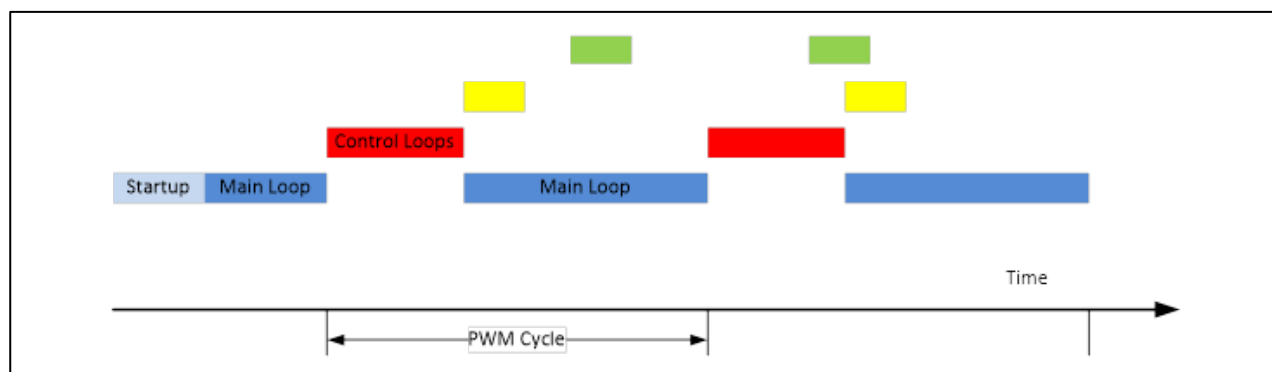


Figure 4.2 Scheduling of firmware tasks (not to scale)

The coordination between the different functional blocks is implemented with the use of shared data structures that encapsulate the information specific for each motor and each communication interface. In the figure below, the arrows represent the data flow between the functional blocks (in rectangles) and the data structures that they share (in ovals).

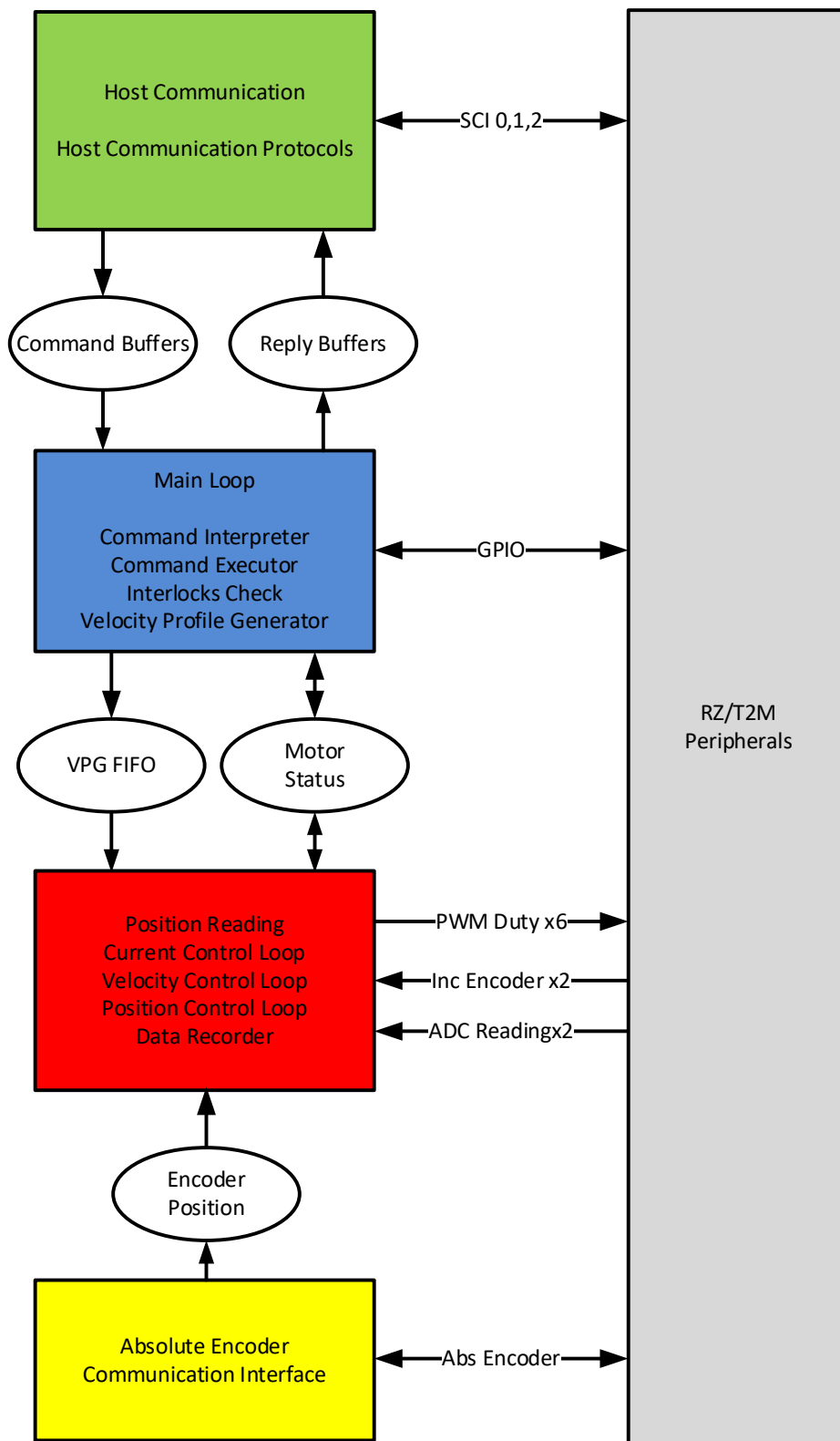


Figure 4.3 Data flow

4.1.1 Startup Functions

The Startup functions are executed upon reset of the device. They are executed only once and are not dependent on any timing constraints. The main phases of the RZ/T2M initialization and the related functions are described below:

Startup Operation	Description
Bootloader	The bootloader executes the first phase of the device initialization responsible for the setting of all clocks and peripherals required for the transferring the firmware executable from the QSPI flash to the device memory.
Peripheral Initialization	The second phase of device initialization invokes the functions that configure the peripherals needed by the motion control firmware. The peripheral initialization are stored in the folder <code>src\cg_src</code> .
Data Structures Initialization <code>m_startup()</code>	The firmware initialization consists of setting up default values to the static data structures, initializing the pointers to the hardware registers to be used by the control algorithms, setting the default state of the GPIOs. The initialization phase completes by loading the programmable encoder protocol configuration based on the selected type. This initialization phase is implemented in the function <code>m_startup()</code> in the <code>src\m_rzt.c</code> file
Persistent Parameters Restore <code>m_Restore()</code>	The firmware has the capability to store the application specific parameters (motor parameters) to the external QSPI Flash memory. At this phase of the initialization, the parameters are restored from the flash and copied to the data structures in RAM. This process avoids the need to recompile the code when a new motor is connected or after control gains are changed to reflect different operating conditions. The motor parameters that can be saved and later restored include motor specific settings (such as motor type and number of pole pairs) encoder specific setting (like type and resolution) and all gains for the current and position control loops.

4.1.2 Non-Real-Time Functions

The non-real time functions are executed from the context of an infinite loop that begins execution after the firmware initialization is completed. The functions executed are described in the table below:

Non-Real-Time Function	Description
Interpret and Execute Host Command m_interpreter()	The interpreter function looks up the host ASCII command in the command dictionary and finds the appropriate command function that needs to be invoked. Alternatively, it directly sets or gets a value when the host command is only requesting parameter change or report.
Run Velocity Profile Generation vpg_update()	The Velocity Profile Generator is invoked periodically to generate set-points required for the execution of a point-to-point motion. Since the execution time of the algorithm depends on the phase of the motion profile, the results are buffered and the real-time task that invokes the position control loop function gets the data through dedicated FIFO buffer. (This function is not supported.)
Trigger Data Recorder m_rec_begin()	This function checks for a condition that will start or stop the data recording functionality. The condition can be selected from a set of options available to the user.
Read Digital Inputs inp_update()	The function reads the data from the digital inputs available on the board and makes them available to the host computer. Any of the inputs can also be configured as a home flag used by the homing sequence.
Check Interlocks interlocks()	The interlocks are conditions reflecting errors that indicate faulty hardware or underperforming actuators. They are evaluated periodically and if any erroneous condition is detected the servo control operation is shutoff.
Run Homing State Machine fsm_homing()	The Homing State machine executes a series of steps that move a servo axis to known position defined by the location where a designated input (home flag) is triggered and / or the position where the encoder index is captured.
Shared Memory Access shared_mem_access()	This function is used to communicate between CPU0 core and CPU1 core when EtherCAT communication is used for motor control. Various data of command or status is sent and received via shared memory.

All of the functions described above are invoked from the function m_foreground() defined in the file m_control.c.

4.1.3 Periodic, Real-Time Functions

The periodic, Real-Time functions are the one executing the motion control algorithms and their determinism is directly dependent on the quality of the motion. For example: any jitter in the position reading timing translates to inaccurate calculation of the derivative component of the position loop affecting the accuracy of the error compensation result.

The real-time operations are executed in the context of the IRQ Handler triggered by the Timer that generates the PWM cycle (50 us).

Real-Time Function	Description
Read Encoder Position <code>pos_read()</code>	The position reading function supports different types of encoder interfaces. In case an incremental encoder is configured this function reads the dedicated timer counter registers. In case an absolute encoder type is selected, the function reads the position from a memory location where the last polling request stored the result. Once the absolute encoder position is obtained, the function initiates another polling transaction so its results will be available for the next time slice. The <code>pos_read()</code> function is implemented in the file <code>m_pos_read.c</code>
Position Control Loop <code>pos_loop()</code>	The position control loop implements a classic PID regulator with enhancements such as Velocity and Acceleration Feed Forward, Output Bias and Output Limit control. The <code>pos_loop()</code> function is implemented in the file <code>m_control.c</code>
Read ADC Values <code>crnt_read()</code>	The current feedback is intended for implementation of the current loop control algorithm as well as the evaluation of the interlock that tracks the motor overload and the amplifier overload. The reading of the ADC values is hardwired to start at the end of each PWM cycle. This is needed in order to eliminate the switching noise impact on the ADC operation. The <code>crnt_read()</code> function is implemented in the file <code>m_rzt.c</code>
Current Control Loop <code>crnt_loop()</code>	The current loop control loop uses the information from the ADC feedback and the encoder position to calculate the direct and quadrature currents creating torque generating magnetic flux. The current PI regulators operation can be disabled, this algorithm also invokes the Space Vector Modulation function that produces the duty cycles for each of the motor phases. The function operates in different modes depending on the selected mode of motor phasing and the phasing status. The <code>crnt_loop()</code> function is implemented in the file <code>m_control.c</code>
Velocity Control Loop <code>vel_loop()</code>	The speed control loop performs PID control from the difference between the target speed and the current speed based on the output result of the position control loop. The <code>vel_loop()</code> function is implemented in the file <code>m_control.c</code>
Recorder <code>m_recorder()</code>	The recorder function is an important feature that enables the real-time data collection for the purpose of tuning motion control parameters or troubleshooting dynamic performance of the firmware. The recorder stores the current values of up to four variables in circular buffers. The start and stop of the data recorder are configurable by the host computer. The <code>m_recorder()</code> function is implemented in the file <code>m_recorder.c</code>

4.1.4 Communication Functions

The first type of communication functions is executed in response to received commands from the host computer. They are invoked by interrupt handler signaling the reception of a new data from the host or signaling ability to send more data to the host computer.

The second type of communication functions handles the interface with the absolute encoders. Typically, they originate request to get the current position periodically.

Communication Function	Description
Data Reception from Host m_rx_interrupt()	<p>The data reception function is serving the interrupt requests from the SCI and handles the specifics of the host communication protocol. This function recognizes the type of the command and the decoding algorithm required. In case of processing binary packet protocols, it also handles the checksum calculation and error handling.</p> <p>This function handles concurrent command requests by maintaining individual buffers for each physical interface. The command interpreter is invoked along with pointer to the command data structure that includes command request and reply buffers. This enables the concurrent support of different host interfaces.</p> <p>The data reception function is implemented in the file m_rzt.c</p>
Data Transmission to Host m_tx_interrupt()	<p>The Data Transmission function communicates the result of the command request back to the host. It utilizes the SCI FIFO buffers to minimize the CPU participation in the communication task.</p> <p>The data transmission function is implemented in the file m_rzt.c</p>
Polling Absolute Encoder Data	<p>The encoder polling is intended to provide up to date position feedback to the control algorithms. This operation is facilitated by dedicated and configurable hardware block in the RZ/T2M and does not involve the main CPU.</p> <p>The polling is initiated by the real-time control task, The result of the encoder position polling is stored in a shared memory in order to be used on the next time slice.</p> <p>The encoder interface functions are implemented in the files with the names corresponding to the encoder communication protocol under the folder src\encoder</p>

4.2 Data Types

This firmware defines all data types in the file **m_common.h**. The table below describes the most important data types and their purpose:

Data type	Description
TReg32	This type represents 32-bit value as individually accessible two 16-bit values and as four 8-bit values.
TMotionParams	This type encapsulates all motion parameters required for the definition of a point-to-point motion such as target position, velocity, acceleration, jerk and velocity profile mode
TMotionProfile	This type keeps the motion profile state at specific time and includes snapshot of the velocity profile generation state, position, velocity, acceleration and stopping distance.
TPosVel	This data structure keeps a pair of position and velocity used by the streaming of velocity profile from the host computer (PVT streaming)
t_motor_pars	This type combines all persistent motor parameters. It is used to maintain a copy of a data structure that can be stored to the Flash memory and restored later.
t_motor	This is main data structure used by the firmware to access all persistent and run-time parameters required for the control of one servo motor. It simplifies the control of multiple motors by instantiating multiple data structure of this type. All motion control functions operate on this data structure by receiving a pointer to specific motor instance as a first parameter.
t_trace	This data structure encapsulates all settings that control the operation of the data recording algorithms.
t_console	This data structure is intended to encapsulate the communication link between the host computer and each communication interface on the Motor Solution Board. This data structure includes command buffer, reply buffer and the pointers that the interrupt handlers use to access the buffers.
t_command	This data type is defined in the m_interpreter.c file where the commands are defined, and the interpreter code is implemented. The structure associates pointer to variable or function with the name of the ASCII command used to expose them to the host computer.

4.3 Data Structures and Variables

The data structures used by the firmware are instances of the types described in the previous chapter. The table below describes the specific variable instances and their purpose in the firmware:

Data structure instance	Description
t_motor m1, m2	These data structures represent the specific settings for each of the two servo motors supported by the Solution Kit firmware.
t_console con0, con1, con2	These variables are dedicated for each of the communication interfaces providing serial connection to a host computer.
short g_counter	This variable retains the most recent value of the PWM Timer counter at the end of the real-time control algorithms operation. The value useful to monitor the CPU utilization for real-time tasks.
long g_tick	This variable is incremented every time slice. It is used to coordinate the operation between the real-time tasks and the main loop.
short g_suspend	This flag is intended for temporary preventing the real-time functions from executing. Its purpose is to enable time-sensitive operations such as writing flash memory from being affected by the real-time functions.
t_command Commands[]	This is an array of command data structures where all host commands are defined. They consist of ASCII name, type and pointer to either function that executes the command or variable that holds the referenced parameter.

4.4 Enumerations, Macros and Constants

The table below lists all enumerations defined in the Solution Kit firmware and the description of the individual values:

Enumeration	Description
ETYPE	Defines the different encoder types supported
VPG_STATE	Defines the states of the Velocity Profile Generator (completed, acceleration, deceleration, plateau, etc.)
VPG_MODE	Defines the different VPG modes – trapezoidal, spline, Bezier
HOMING_STATES	Defines the state machine of the Homing algorithm
CommutationModes	Defines the different operation of the current loop algorithm
ParserStates	Defines the states of the binary protocol parser
ProtocolTypeRequest	Defines the type of host message (ASCII, Packet)
PacketCode	Defines the type of the packet received
PacketError	Defines the possible errors reported by the packet protocol interpreter

5. Initialization and Startup Functions

5.1 Bootloader

The bootloader transfers executable code from the QSPI flash memory to the RZ/T2M's ATCM. Once the data transfer is complete, execution flow proceeds to the `hal_entry()` function defined in the `hal_entry` file.

The file that implements this function is stored in the project folder `rzt\fsp\src\bsp\cmsis\Device\RENESAS\Source\startup.c`.

5.2 Peripherals Initialization

The source files that initialize all peripheral functions used in this firmware are shown below.:

File Name	Peripheral Description
<code>\cg_src\r_cg_mtu3.c</code>	Configures MTU timers as position decoders for the Incremental Encoder phases
<code>\cg_src\r_cg_poe3.c</code>	Configures the POE3 unit for overcurrent detection
<code>\cg_src\r_cg_port.c</code>	Configures the state, mode and the strength of the GPIO pins
<code>\cg_src\r_cg_s12ad.c</code>	Configures the ADC for the current feedback
<code>\cg_src\r_cg_scifa.c</code>	Configures the SCI communication interfaces
<code>\src\drv\dsm\r_dsmif.c</code>	Configures the DSMIF for the current feedback

5.3 Firmware Initialization

The firmware initialization is intended to start the operation of the peripherals, initialize the internal data structures, restore the motor parameters from the persistent storage and configure the programmable encoder interface block. This sequence is implemented by the function **m_startup()** defined in the file **m_rzt.c**.

The firmware defines two instances of the data structure that defines all motor specific parameters – m1 and m2. First, the startup function initializes the members to point at the registers of the timers assigned to each individual channel. The use of pointers to the different timer channels enables the sharing of the functions that operate on them between the two channels. They do not have to be concerned with the specific addresses of each timer.

Next, the startup algorithm invokes the function **setup_motor()** for each of the two motors. This function sets the default parameters of all motor structure data members. Then it invokes the function **m_Restore()** to read the motor parameters saved to the QSPI flash memory.

Once the data structures are initialized, the startup function starts the timers generating PWM output - **R_MTU3_C3_4_6_7_Start()** and the timers processing the incremental encoder counting - **R_MTU3_C1_Start()** and **R_MTU3_C2_Start()**.

The function **setup_scif()** is invoked to configure the data send/receive buffer pointers for each SCI interface as well as the data structures (con2) of type **t_console**, that enable their independent operation. Finally, the SCI interfaces are enabled by invocation of the functions **R_SCIFAx_Start()**.

The startup procedure outputs the character “R” to the SCI4 interface signaling any host connected to the serial interface, that the module is ready.

Next the startup initializes the data structures required for supporting the data recording operation. This is done by invoking the function **m_TraceSetup()**.

As a last step of the startup function, the algorithm initializes the configurable encoder interface hardware. This operation is conditional on the encoder type set to the motor data structures. If it is not incremental type, then the algorithm waits for 150ms and then invokes the function **setup_encoder()** for each of the motors.

With this the firmware initialization is completed and the execution flow is returned to the main loop that repeatedly invokes the function **m_foreground()**.

6. Servo Control Operation

The servo control loop is designed to maintain a desired motor position. This is accomplished by periodically evaluating the difference between the desired and the current position and calculating a compensation to be executed by the motor.

When the servo control loop is first enabled, the desired position is set equal to the current position of the motor. In this state, the algorithm maintains the current position in one place by compensating external disturbances such as gravity, voltage sag or mechanical forces.

When a motion command is issued, a dedicated algorithm (Velocity Profile Generator) calculates the desired of position for each time the servo control loop is executed. The ability of the servo control loop to follow the desired positions creates a motion with programmed and velocity and acceleration.

The operation of the servo control loop and the main functions taking part of it is shown on the figure below:

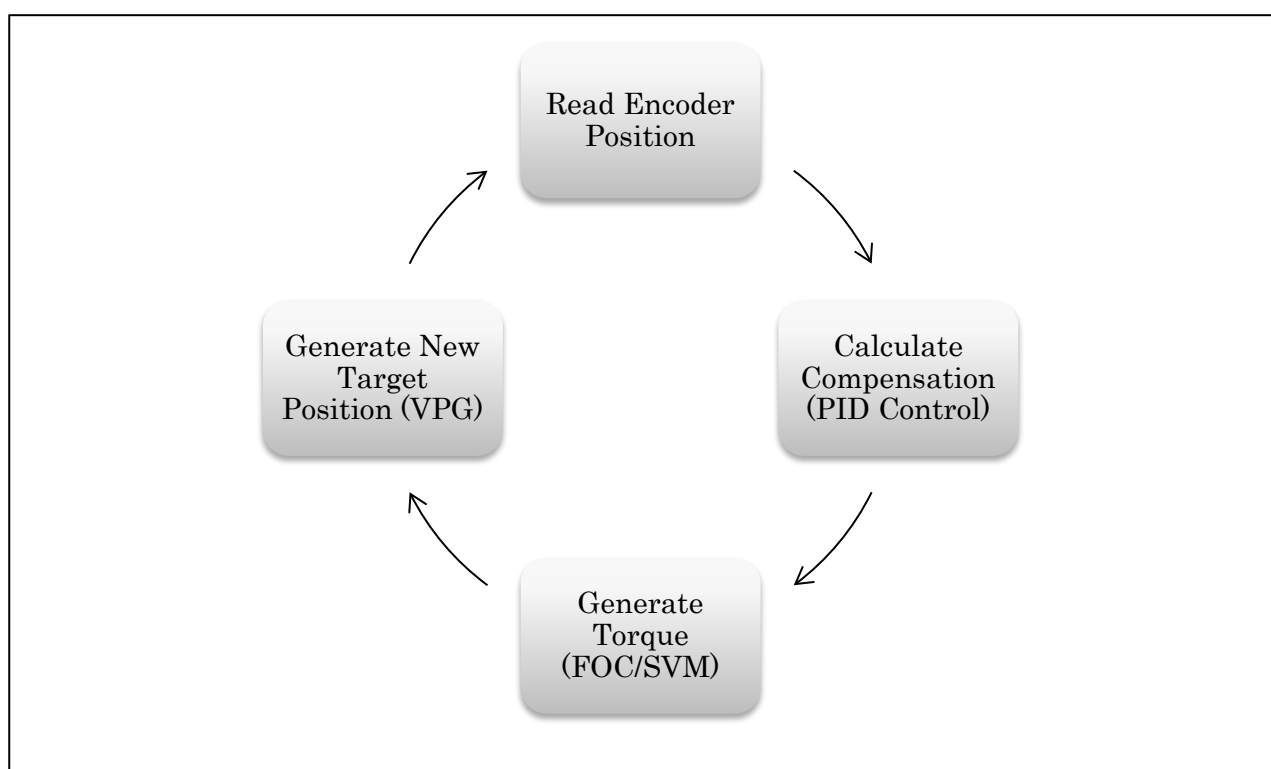


Figure 6.1 Servo Control Loop

The servo control loop is executed continuously with programmable period. The accuracy of the time between each run is critical for the accuracy of the position error compensation calculations and the ability of the controller to minimize the settle time at the end of any motion.

6.1 Motor Position - Encoder Interface

The purpose of the position feedback is to provide accurate information about the motor position. It is used as an input parameter to the motor control commutation as well as to the position control described below.

The Solution Kit Firmware includes variety of encoder feedback options. The possible encoder options are described in the enumeration **ETYPE**. The currently configured encoder type is stored in the **t_motor** data member **encoder_type**. The table below describes the valid settings for this variable:

ETYPE Enumeration	Encoder Type
ETYPE_INCREMENTAL = 0	Incremental encoder – the position change is represented by two phased pulse sequence where the phase between them indicates the direction of motion. A dedicated mode of the timer operation is decoding direction and counting the pulses representing position change. The software reads the current position from the timer counter. It has 16-bit resolution, so the software expands the range to 32 bits.
ETYPE_APE_ENDAT = 1	Absolute encoder with EnDat 2.2 communication protocol.
ETYPE_APE_BISS = 2	Absolute encoder with BiSS communication protocol.
ETYPE_APE_FACODER = 3	Absolute encoder with FA-Coder communication protocol.
ETYPE_APE_AFORMAT = 4	Absolute encoder with A-format communication protocol.
ETYPE_APE_HIPERFACE_DSL = 5	Absolute encoder with Hiperface DSL communication protocol

The incremental encoder feedback position is lost after power cycle and this requires the execution of algorithms such as Phasing for rotor position identification and Homing – for machine position identification. Another incremental encoder specific feature is the presence of Index pulse - once every revolution. The RZ/T2M timer is capable of using the encoder index as a trigger to capture the position where it has occurred. This operation is hardware defined and does not depend on the speed of motor rotation or any software latency. The index capture mechanism is used for precise initialization of the motor coordinate system as part of the homing procedure.

The absolute encoders eliminate the need for phasing and homing, but require additional configuration parameters (bit rate, bus delay compensation). Depending on the absolute encoder technology they also may require battery backed multi-turn counter and the software have to monitor their status for possible battery fault condition. The RZ/T2M blocks perform all interaction with the absolute encoders without the need of CPU involvement.

The absolute encoders offer the ability to store application specific information in an internal EEPROM memory. The Solution Kit firmware includes methods that access the EEPROM memory of the connected absolute encoder.

Important Notice!

The initialization of the RZ/T2M encoder interface hardware can only be executed once per power cycle. For this reason, the change from one absolute encoder type to another requires intermediate switch to incremental encoder type. Example of switching from BiSS to EnDat encoder:

(Assuming the current encoder type is ETYPE_APE_BISS)

1. Set the encoder to ETYPE_INCREMENTAL
2. Save the motion parameters to Flash memory
3. Restart firmware
4. Set the encoder to ETYPE_APE_ENDAT
5. Save the motion parameters to QSPI Flash memory

The change between absolute and incremental encoder does not require power cycle.

6.2 Motor Control - Torque Generator

This firmware supports Brushless DC Motors. The type of the motor being controlled is defined by the motor control structure data member **motor_type**. The valid settings are described in the table below:

motor_type setting	Motor Type
3	BLDC / PMSM 3-Phase Motor

The Brushless DC (BLDC) motor category includes Permanent Magnet Synchronous Motors (PMSM) as well as Linear Motors and AC Servo Motors. All of them share the same principle of torque generation created by three-phase stator windings and permanent magnet rotor with different number of pole pairs.

The three-phase motors are controlled by creating a three-phase voltage that produces magnetic flux with desired magnitude and orientation. This flux can be represented as Space Vector and the process of its calculation is called Space Vector Modulation. It takes the two flux components – magnitude and angle and produces three phase PWM duty cycle numbers stored in the Timer registers to generate the designed phase voltage.

The input value for the torque generating algorithm can be interpreted as desired voltage or current. The use of input voltage is simple because the only information required for its calculation is the motor position. This approach can be called open loop voltage control – it is shown on the picture below:

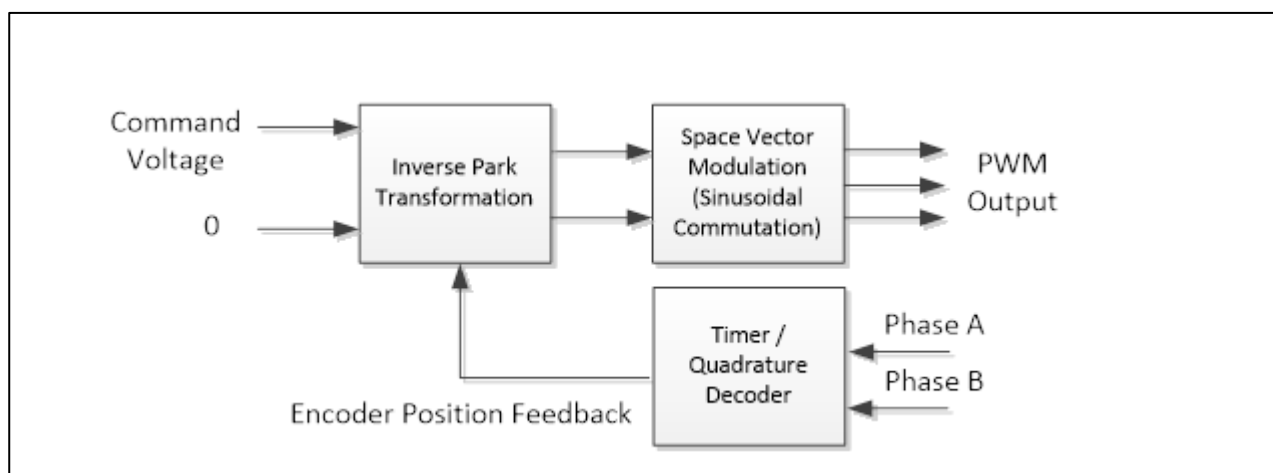


Figure 6.2 Encoder Based Voltage Control

The voltage control does not reflect the actual torque created due to the fact that every motor produces Back Electromotive Force (BEMF). The BEMF voltage is proportional to the speed of the motor and the number of the windings of each phase. As a result, the effective voltage applied to the motor windings is the difference between the controller PWM output voltage and the BEMF generated voltage. Subsequently, the current flowing through the motor windings is dependent on the speed of the motor.

The Field Oriented Control (FOC) algorithm is developed to eliminate the significance of the motor speed to the generated torque. This is accomplished by implementing current feedback that provides measure for the actual current (and torque) being produced. The FOC algorithm structure is described in the diagram below:

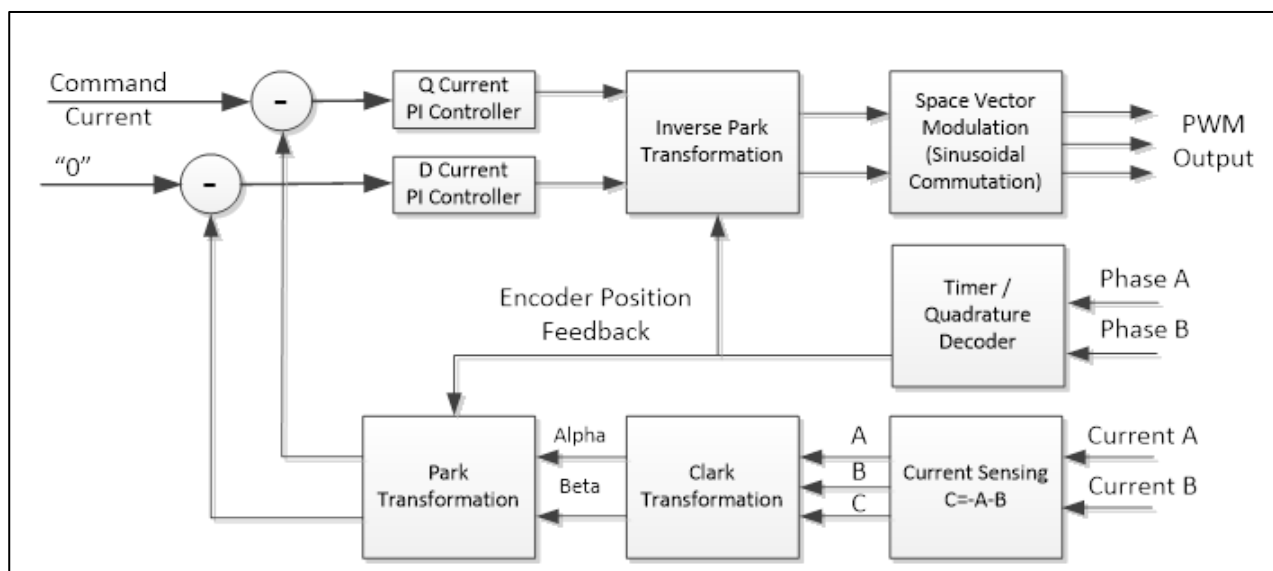


Figure 6.3 Field Oriented Control Structure

The currents of two of the three phases are sampled continuously by the controller ADCs. The third phase current is reconstructed ($I_a + I_b + I_c = 0$). The Clark and Park transformations are calculating the components of the torque vector as two orthogonal vectors of representing two current currents:

- Quadrature current – it represents the torque generating flux, perpendicular to the N-S poles of the rotor.
- Direct current – it represents the heat generating force – it should be 0 at all times.

Each of the two currents is compared with its set points and the error is compensated by Proportional – Integral (PI) regulators. The described FOC algorithm is implemented by the function `commutate_foc()` in the file `m_control.c`. The algorithm uses the following motor parameters as input values:

t_motor data member	Description
counts2rad	Coefficient representing the resolution of the encoder per one electrical cycle.
phase_angle	This parameter represents the orientation of the flux vector with respect to the rotor North pole.
p_iu	Pointer to the variable holding the U-phase sampled current
p_iv	Pointer to the variable holding the V-phase sampled current
foc_id	Calculated direct current
foc_iq	Calculated quadrature current
foc_id_err	Calculated direct current error
foc_iq_err	Calculated quadrature current error
foc_vd	Calculated direct voltage
foc_vq	Calculated quadrature voltage
foc_alpha	Alpha component of the voltage vector
foc_beta	Beta component of the voltage vector

The final step of the motor control algorithm execution is the space vector modulation. It is implemented by the function **commutate_svm()** in the file **m_commutation.c**. This function takes the Alpha and Beta voltages calculated by the FOC algorithm and returns the corresponding duty cycle for each phase.

The algorithm for torque generation is configurable run-time by the setting of the motor variable **commutation_mode**.

commutation_mode setting	Description
3	External / User Defined Phase Voltage Setting

6.3 Position Control – PID Regulator

The position control algorithm takes the position error as an input and calculates output result intended to counter this error. The position loop control function is called `pid_calc()` and is implemented in the file `m_pid_calc.c`.

The structure of the PID regulator is shown on the picture below:

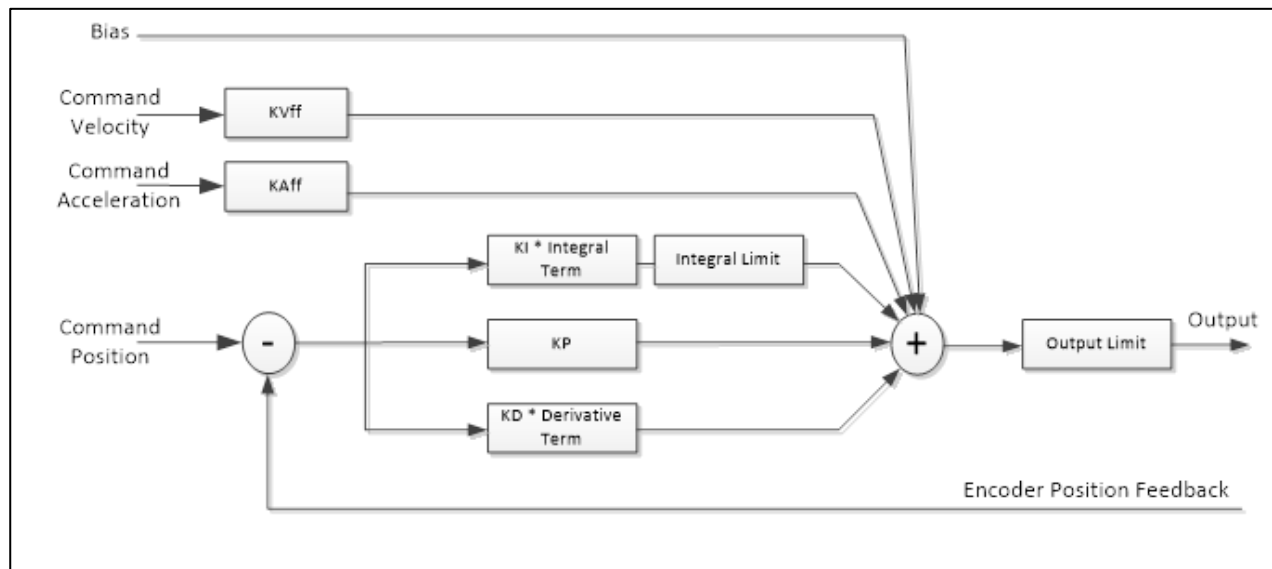


Figure 6.4 PID Regulator Structure

The position error is calculated and passed as an input parameter to the `pid_calc()` function. The position error is represented in encoder counts.

The configuration parameters for the position control function are stored in the `t_motor` data structure. They are described in the detail in the table below:

t_motor Data Member	Description
<code>crnt_kp</code>	Proportional gain (0 – 32767)
<code>crnt_ki</code>	Integral gain (0 – 32767)
<code>crnt_kd</code>	Differential gain (0 – 32767)
<code>integral_limit</code>	Integration limit (0 – 32767)
<code>crnt_kvff</code>	Velocity feed forward (velocity gain)
<code>crnt_kaff</code>	Acceleration feed forward (acceleration gain)
<code>crnt_bias</code>	Bias – added directly to the output result
<code>cmd_vel</code>	Command velocity
<code>cmd_acc</code>	Command acceleration
<code>pos_loop_limit</code>	Output limit (0 – 32767)
Temporary variables:	
<code>pos_error</code>	Stores the position error from the last invocation of the <code>pid_calc()</code> function.
<code>derivative_err</code>	Stores the calculated derivative of the position error (the difference between the last and the current position error)
<code>integral_err</code>	Stores the calculated integral of position error

The PID control parameters are not directly accessible by the host computer. They are buffered and copied to the variables used by the `pid_calc()` function only when the servo control is turned on, a new motion is started or an existing motion is stopped. The parameters are updated inside the function `m_update()` that gets invoked by the functions described above. The picture below shows the parameter buffering mechanism and the related functions.

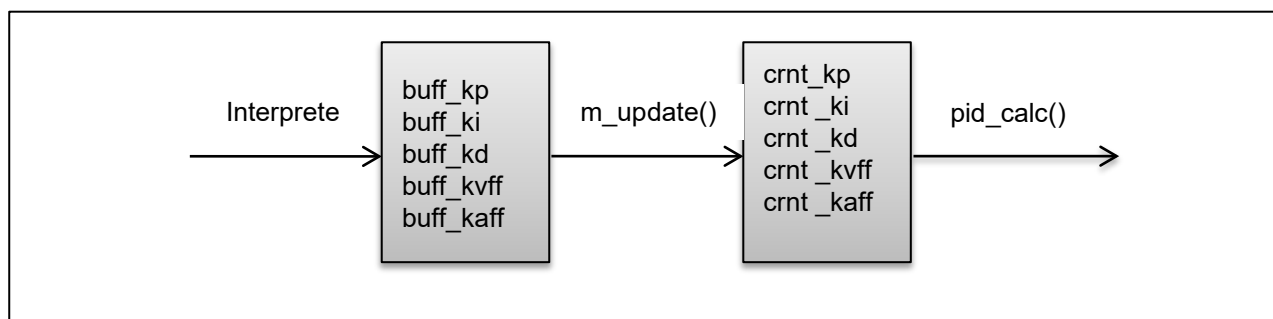


Figure 6.5 Buffering of Position Loop Parameters

The purpose of the PID gains buffering is intended to update them synchronously and all at the same time. Setting the parameters one at a time or allowing the update to be interrupted may lead to glitch in the output of the control function. This would introduce undesired position error.

The function `pid_calc()` returns a new output value to be fed as an input to the motor commutation algorithms described in the previous chapter. The output value range is in the range of ± 32767 .

6.4 Motion Planning - Velocity Profile Generator (Not supported)

The Motion Planning function defines how the motor will execute a motion from one position to another. First, the algorithm uses the current and the target positions to calculate the desired travel distance. Then it uses the defined maximum acceleration, velocity and deceleration to calculate the length of the acceleration and deceleration motion phases. The figure below presents typical trapezoidal velocity profile that includes all motion phases:

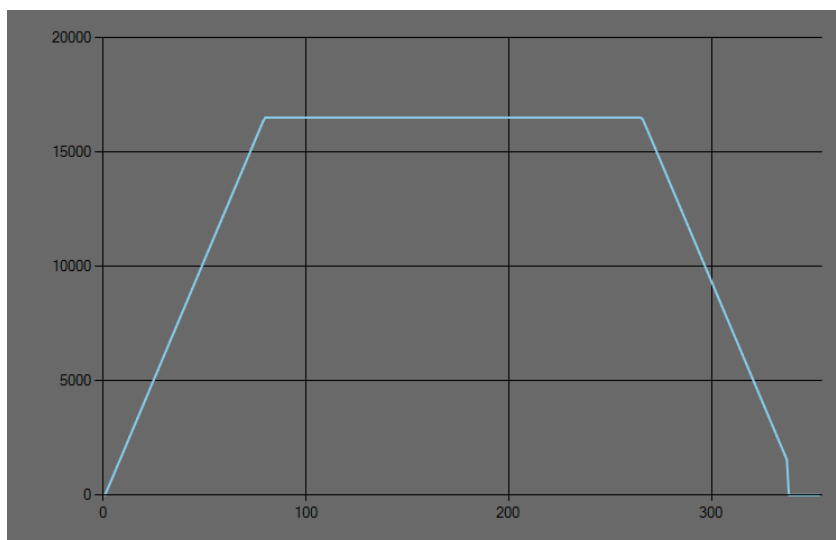


Figure 6.6 Trapezoidal Velocity Profile

Based on the travel distance and the motion parameters, the algorithm also determines the length of the constant velocity motion phase. In case the travel distance is too short, the motion may never reach the maximum desired velocity.

The diagrams below present an example of a short move where the motion never reaches the maximum velocity.

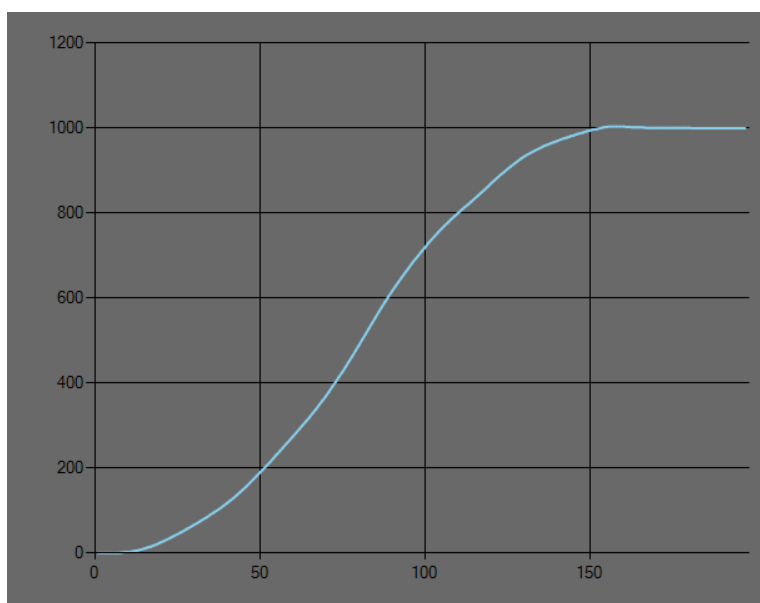


Figure 6.7 Position Profile for short move to 1000

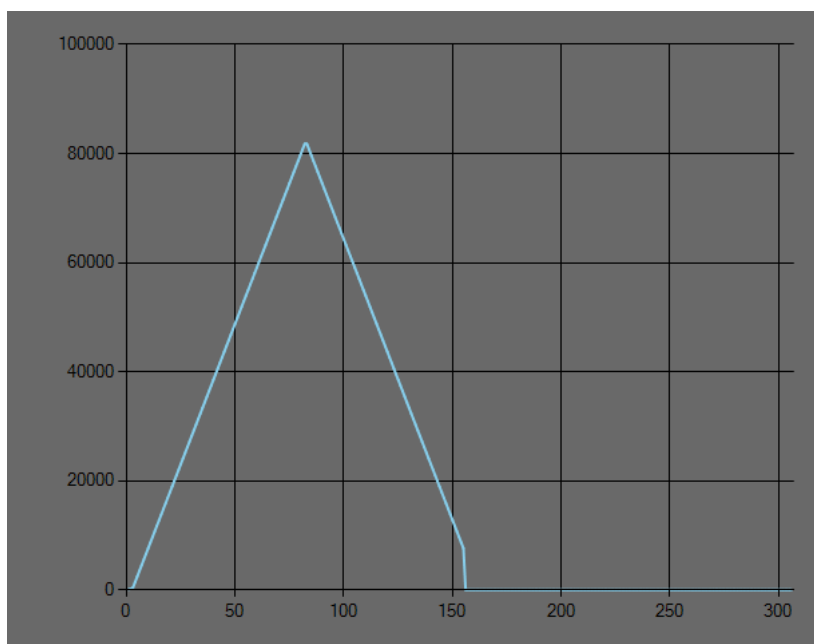


Figure 6.8 Trapezoidal Profile for a short move

This firmware includes several velocity profile generation algorithms that use different math functions to shape the velocity profile such that it will meet the application specific requirements. The figure below shows the velocity profile calculated with the help of a Spline function:

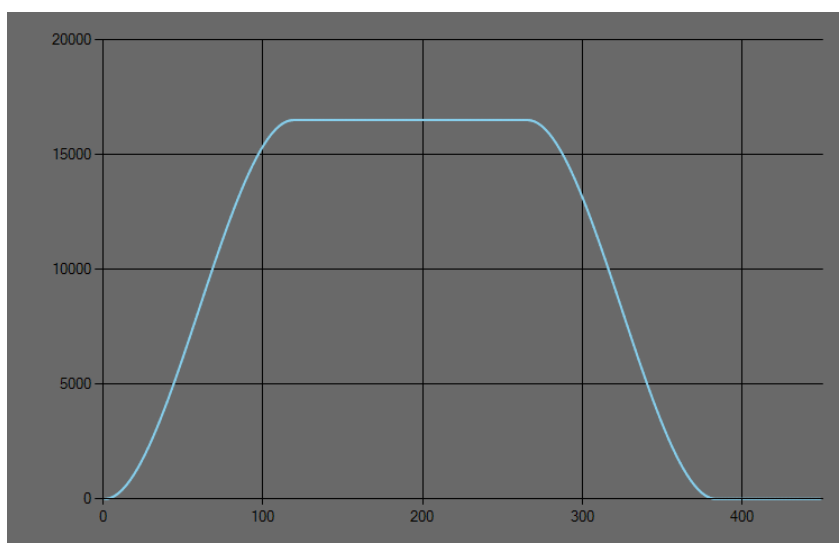


Figure 6.9 Spline-based Velocity Profile

The smoother velocity profile reduces the vibrations inherent to the trapezoidal profile, but this comes at the expense of longer time to reach a target position. The selection of specific velocity profile is a tradeoff between the time to reach a target position and the settle time at the end of the motion.

The figure below shows a velocity profile generated with the help of a Bezier-curve which enables the individual definition not only of the acceleration and deceleration settings, but of their derivatives (jerk) as well.

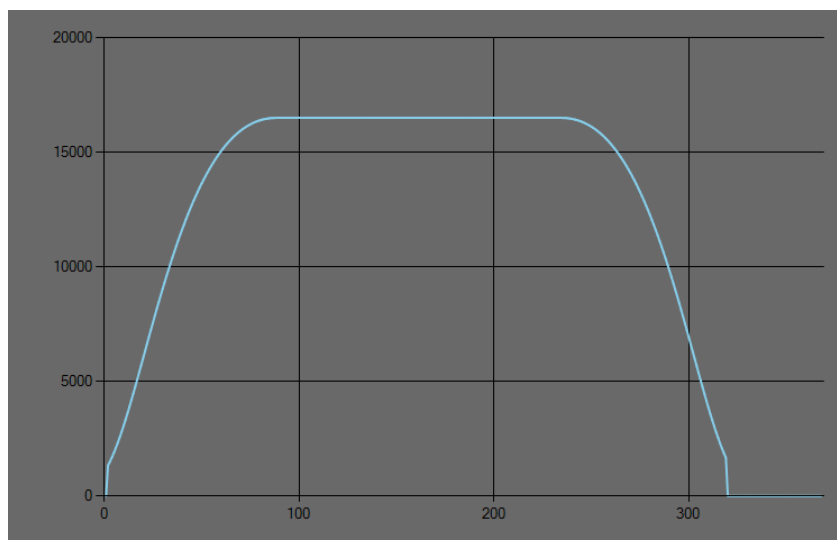


Figure 6.10 Bezier-curve based Velocity Profile

All of the velocity profiles described above only execute a single motion from one position to another. The velocity is zero at the beginning and at the end of this operation. This mechanism is not sufficient for the control of complex mechanism such as robots, gantry stages or CNC machines. These applications require that the host computer generates the complex velocity profiles of all motors. Since the communication bandwidth between the host and the controller is inherently limited, the velocity profiles are presented as sets of Position and Velocity over a fixed time slices (typically 5ms to 20ms). Hence the name Position-Velocity-Time for these profile time. The PVT points are streamed to each of the controllers which in turn execute interpolation algorithm to generate the desired position and velocity set-points each 50 microseconds.

The Velocity Profile Generator operation is defined by state machine with the following states:

State	Value	Description
Idle / Motion Completed	0	Default / Final state.
Acceleration	1	The motion is accelerating
Deceleration	2	The motion is decelerating
Plateau	3	Constant velocity
Streaming / PVT Interpolation	4	Interpolation

The state of the velocity profile generator is stored in the `t_motor` data member `vpg_state`.

When the motion is started all motion preparation is handled by the function `update_ctrl()`. It in turn invokes the startup function of the currently selected velocity profile generator that returns the new VPG state. From this point on the VPG is invoked periodically to produce position and velocity set-points for every cycle of the position control loop. In addition, the VPG returns the new state. Once the motion profile is completed, the state is set to Idle and the periodic VPG function is not invoked anymore.

The table below describes the startup and the periodically invoked functions for each velocity profile type:

VPG Type	Startup VPG Function	Periodic VPG Function
	update_ctrl()	vpg_update()
Trapezoidal	vpg_trap_start()	vpg_trap_next()
Spline-Curve	vpg_spline_start()	vpg_spline_next()
Bezier-Curve	vpg_bezier_start()	vpg_bezier_next()
PVT Interpolation	vpg_pvt_start()	vpg_pvt_next()

The data produced by the VPG is not passed directly to the Position Control Loop. Instead they are buffered in a dedicated FIFO buffer. This mechanism allows the asynchronous execution of the VPG functions with respect to the position control loop operation. This separation serves two goals:

1. The generation of the velocity profile can be very complex and may require calculation ahead of time. The asynchronous approach avoids the need to “oversize” computational budget of the real-time position and current control loops.
2. The separation of the VPG calculation from the real-time control loop allows for easy redesign where the VPG generator is running on another CPU core or even on another network device.

6.5 Motion Control Parameters (Not Supported)

The Motion Control Parameters include the settings that define where the motor is going to and how it is supposed to get there. The motion to be executed also depends on the current state of the motion controller – its current position, position error, current velocity and current state.

6.5.1 Target Position

The target position is expressed in encoder counts. It can be defined explicitly with a request for Absolute Target Position. (ASCII command ABS).

Alternatively, the target position can be defined as a distance relative to the current position (ASCII command REL). This way of defining the target obviously depends on the motor position at the moment the motion is started. Note that the between specifying the relative distance and the moment the motion is started, the current position may change. This in turn will lead to change of the expected target position.

6.5.2 Maximum Velocity

The maximum velocity parameter defines a limit that may or may not be reached depending on the other motion parameters. The velocity parameter units are defined as “Encoder Counts per Position Loop interval”. Since the position loop interval can be very short time (as little as 50 microseconds), the velocity value is communicated as a fixed point number in 16.16 bit format after being multiplied by 65536.

The conversion of the units from Encoder Counts per Second to the Solution Kit firmware format, the number has to be multiplied by the position loop time slice and then multiplied by 65536. For example:

If the position loop is running at 100 microseconds then 5000 enc.counts per second is converted as:

$$5000 * 0.0001 * 65536 = 32768$$

The maximum velocity value corresponding to 5000 enc.counts per second is 32768

6.5.3 Maximum Acceleration and Deceleration

The maximum acceleration and deceleration parameters define a velocity profile slope that may or may not be reached depending on the capabilities of the control hardware and the specific motor load. The acceleration parameter units are defined as “Encoder Counts per Position Loop interval squared”. Since the

position loop interval can be very short time (as little as 50 microseconds), the acceleration and deceleration values are communicated as a fixed point number in 16.16 bit format after being multiplied by 65536.

The conversion of the units from Encoder Counts per Second to the Solution Kit firmware format, the number has to be multiplied by the position loop time slice squared and then multiplied by 65536. For example:

If the position loop is running at 100 microseconds then 30000 enc.counts per second squared is converted as:

$$30000 * 0.0001 * 0.0001 * 65536 = 19$$

The maximum acceleration value corresponding to 30000 enc.counts per second is 19

6.5.4 Maximum Acceleration and Deceleration Jerk

The maximum acceleration and deceleration jerk parameters define a velocity profile slope only when a Bezier-curve velocity profile is being used. The jerk units are dimensionless because they define the Bezier-curve tangent orientation as a ratio. The Jerk value is expressed as integer between 0 and 1000.

6.5.5 Motion Start Modes

The point-to-point motion is started with a single function (ASCII command GO). It invokes the function `update_ctrl()` which disables the interrupts and copies all motion parameters and position loop parameters from their buffered locations to the variables used by the control loop algorithms. This is intended to update all control parameters simultaneously because of their interdependencies.

The trapezoidal velocity profile allows change of its motion parameters on-the-fly. This allows a new motion to be started even before the last one is completed. The new motion may have some or all of its parameters changed.

Alternatively, the PVT streaming mode can initiate a motion after the PVT buffer is filled to a certain level.

6.5.6 Motion Stop Modes

The motion stop modes allow the motion to be stopped in a controlled manner even if the target position is not reached. There are three modes of stopping available:

1. Smooth Stop - in this mode the controller executes graceful completion of the current motion. This is accomplished by switching the current velocity profile generator to Trapezoidal and calculating target position based on the programmed maximum deceleration.
2. Abrupt Stop – in this mode the controller uses the deceleration 32 times the maximum defined value. This mode is intended for emergency situations.
3. Servo Off Stop – in this mode the servo control is turned off and the motor windings are shorted by the inverter so that it operates in dynamic braking mode – using the Back EMF as a stopping force.

7. System Control Functions

7.1 Interlocks

The interlocks are hardware defined or software calculated conditions indicating abnormal state of the control system. They are implemented in the function `interlocks()` which is in the file `m_interlock.c`.

The figure below summarizes the algorithm implemented by the `interlocks()` function.

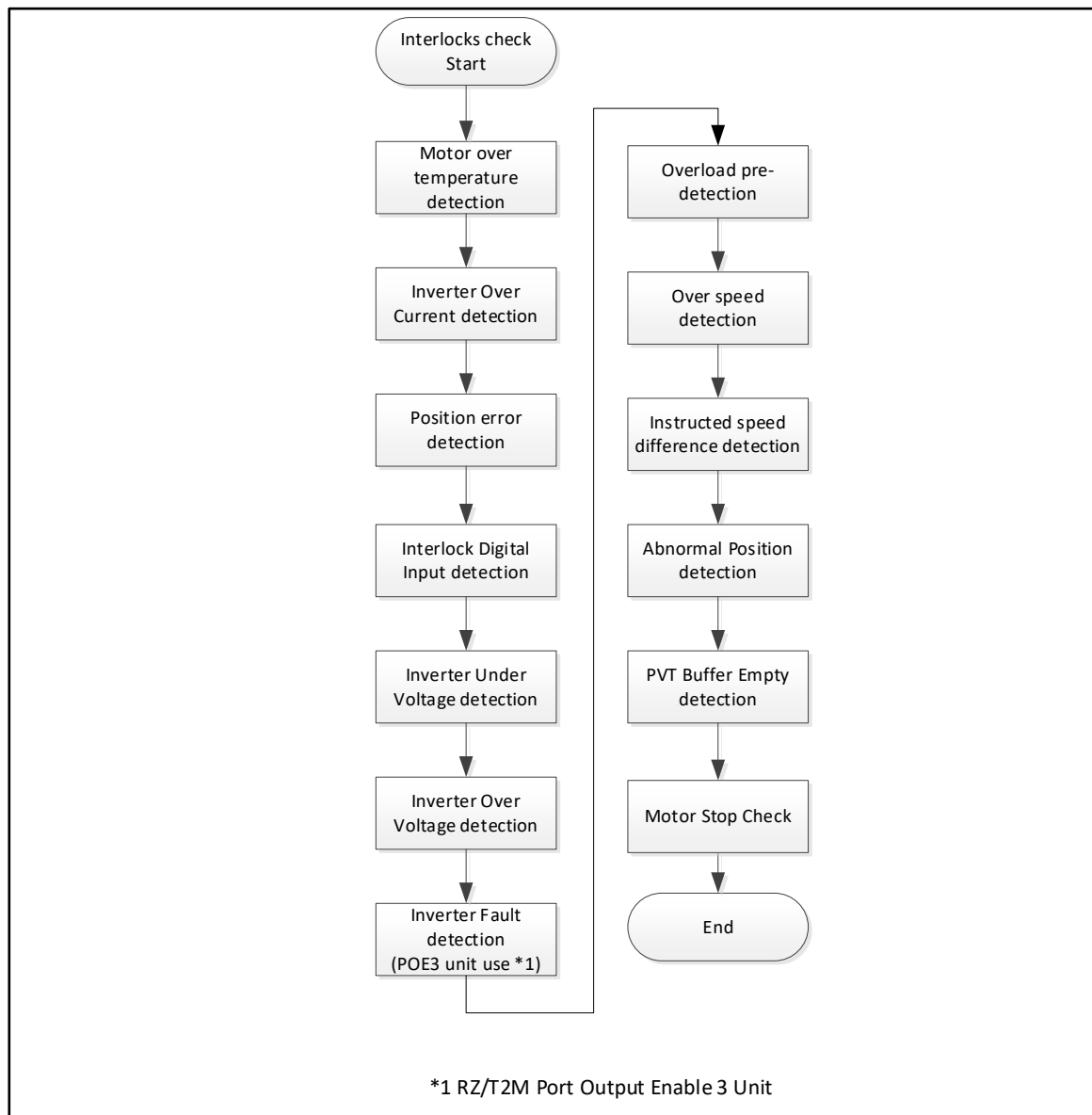


Figure 7.1 Interlocks Evaluation

1. Motor over temperature detection

The first interlock is tracking the energy consumed and compares it with the threshold value reflecting the ability of the motor to dissipate heat resulting from its efficiency. The algorithm calculates the square of the measured current consumption. Then it subtracts the nominal current consumption parameter (I2t_nominal). The result is integrated in the variable I2t_integral and then the integral is compared to configurable limit (I2t_limit). Since the overheating is a relatively slow process, when the limit is exceeded the only action is raising a flag in the activity state (ACT_MtOverTemp). When the integral value drops below the limit, the flag is cleared.

2. Inverter Over Current detection

The next interlock is comparing the total current consumption to a limit (tc_limit) that is exceeded more than a certain time (tc_limit_time). The overcurrent interlock is intended to prevent overloading the motor or the machine connected to it in case of mechanical obstruction or other disturbances. When the overload condition is detected, the interlock function sets a flag in the motor activity state (ACT_OverCurrent) and turns off the servo control loop. Subsequently, any motion is stopped as well.

3. Position error detection

The Position Error interlock tracks the absolute value of the position error. If the position error exceeds the limit (pos_error_limit) then a dedicated timer (pos_error_timer) starts measuring the duration of the error condition. If it exceeds the configured time (pos_error_time), the interlock function stops any motion. Depending on the value of the variable auto_stop_mode it may also turn off the servo control. This interlock condition is indicated in the motor activity state by raising the flag ACT_PosError.

4. Interlock Digital Input detection

The interlock function can be configured to treat any of the available digital inputs as triggers for an Interlock condition. The inputs to be evaluated are defined as a bit-mask in the variable inputs_err_mask. The triggering of the interlock is instantaneous after the AND operation between the inputs byte and the mask is evaluated is non-zero. In response the interlock function turns off the servo control and the power amplifier output is disabled. The motor activity state is also updated by raising the flag ACT_Inhibit.

5. Inverter Under Voltage detection

Set the error detection flag when the inverter bus voltage falls below the threshold value.

6. Inverter Over Voltage detection

Set the error detection flag when the inverter bus voltage exceeds the threshold value.

7. Inverter Fault detection (POE3 unit use)

The Fault signal of the inverter board is monitored with POE3 Unit, the PWM output is automatically switched to high impedance when the Fault signal is detected, and the error detection flag is set.

8. Overload pre-detection

Set the error detection flag when the inverter current value exceeds the threshold value. By setting a lower threshold than "Inverter Overcurrent detection" and masking the servo OFF in this item, it is possible to detect the alarm before the servo turns off due to overcurrent.

9. Over speed detection

Set the error detection flag when the motor speed exceeds the threshold value.

10. Instructed speed difference detection

Set the error detection flag when the change in motor rotation speed exceeds the threshold for 5 seconds.

11. Abnormal Position detection

Set the error detection flag when the position information of the motor deviates from the upper limit threshold or the lower limit threshold range.

12. PVT Buffer Empty detection

Checks the FIFO state of the Velocity Profile Generator every 100μs and sets the error detection flag when the state of Empty exceeds the threshold number of times.

13. Motor Stop Check

Checks the status of each error detection flag and mask setting and stops the motor when the condition is satisfied.

7.2 Electronic Gearing(Non Supported)

The Electronic Gearing provides synchronization (coupling) of the operation of two independent motors by software means. The Electronic Gear function is a linear / proportion relationship between the desired positions of two motors.

When the electronic gearing is active, one of the motors (the master) is given target position commands. The command position of the second motor (the slave) is derived from the master using multiplication by the electronic gear ratio coefficient. Every command that controls the servo status, start and stop of the motion also acts on both motors simultaneously.

The electronic gear ratio is defined by two motor variables: gear_in and gear_out. They act as a nominator and a denominator in the calculation of the electronic gearing coefficient.

The update of the slave axes set position, velocity and acceleration is implemented in the function set_slave() defined in the file m_control.c. This function is invoked by the servo control loop if the motor is designated as Electronic Gearing Master. This role is selected by the variable module_type when it is set to TYPE_EGEAR.

7.3 Digital Inputs and Outputs

The firmware continues to poll the status of digital input.

Digital input is notified to the host of 8bit of the motor solution board SW0501. The following is a list of SW0501.

The host can be notified by the GPIO command. The GPIO command is executed with the function `m_gpio ()` and notifies the 16 -bit value. (The top 8bit always returns 0.)

Bit#	b7	b6	b5	b4	b3	b2	b1	b0
Name	CFG7	CFG6	CFG5	CFG4	CFG3	CFG2	CFG1	CFG0
Port	P17_1	P22_3	P11_2	P07_6	P01_1	P11_5	P10_7	P10_6

7.4 Data Recording

The Data Recording functions are intended to enable the analysis of the system behavior in real time. It is indispensable tool for analyzing the performance of the different control loops, their configuration parameters and their efficiency in application specific contest. The data recorder stores up to four user defined parameters in buffers during system operation. The support functions and configuration parameters enable the selection of rate of recording, which variables are recorded, when the recording is started and when it is supposed to end.

The length of the recording as number of samples is defined by the macro `TRACE_BUFFER_SIZE`. By default it is set to 512. This value can be increased when the application requires longer records and the RAM memory is available. All buffers are combined into a single array named `traceData[]`. The data type of the array is short – 16-bit integer. For this reason, when a 32-bit variable is being recorded, its value is split between the first and the fourth buffers. When the data is reported, it is combined appropriately.

The host specifies the data to be recorded for each channel by using the commands CH1, CH2, CH3 and CH4. These commands are used to set a code representing the data of interest. The first 8 codes are reserved for 32-bit variables. The rest are referencing 16-bit data. The table below defines the correspondence between different data variables and the codes that need to be set for their recording:

CH1, CH2, CH3, CH4 codes	Referenced motor variable
0	Motor position
1	Commanded velocity
2	Commanded acceleration
3	I2t Integral
4 - 7	Reserved
8	Position Error
9	PID Regulator Output Value
10	Reserved
11	Direct Current (heat generating)
12	Quadrature Current (torque generating)
13	Direct Current Error
14	Quadrature Current Error

RZ/T2M Group	RZ/T2M Motor Solution Kit Firmware (Motor Control, EtherCAT)
--------------	--

15	Raw Current A (U Phase A/D value)
16	Raw Current B (V Phase A/D value)
17	PVT FIFO Buffer depth
18	FOC Voltage Output D
19	FOC Voltage Output Q
20	Real time task timing
21	Phase Angle
22	Raw Current C (W Phase A/D value)
23	Raw Position
24	Raw Position Error
25	Position control integral quantity
26	Speed difference value
27	Speed control integral quantity
28	Current control integral quantity (Id)
29	Current control integral quantity (Iq)
30	Motor torque estimate
31	Encoder detection angle (electrical angle)

The start and stop conditions of the data recorder are configured by the ASCII command TRACE (variable trace.Trigger). The table below describes the possible settings representing different trigger conditions:

TRACE codes	Trigger Start Condition	Trigger Stop Condition
0	N/A	Stop data recording
1	Start recording immediately.	Stop when the motion is completed. This trigger is useful for examining the end of a motion.
2	Start recording immediately.	Stop when the buffer is full. This trigger is useful for examining the beginning of a motion.
3	Start recording on start of motion.	Stop on end of motion.
4	Start recording immediately.	Stop on input change. The input bit mask is defined in the trace.Level variable.
5	Start recording immediately.	Stop on value exceeding the threshold defined in trace.Level variable.
6	Start recording immediately.	Stop on value below the threshold defined in trace.Level variable.
7	Start on PWM output change.	Stop when the buffer is full.
8	Start recording on input change. Input mask is defined in the trace.Level variable.	Stop when the buffer is full.
9	Start on value exceeding the threshold defined in trace.Level variable.	Stop when the buffer is full.
10	Start on value below the threshold defined in trace.Level variable.	Stop when the buffer is full.

The recorder operates synchronously to the real-time task that executes every 50us. The rate of the recorder can be expressed as multiples of this time interval. The multiple factor is set by the ASCII command TRATE and stored in the variable trace.RateMult. For example, if the desired rate of the data recorder is 1ms then the TRATE should be set to 20.

Another variable evaluated during some of the trigger conditions is the ASCII command TLEVEL (variable trace.Level). The value of this variable is the threshold that the recorded variable is compared against. Depending on the trigger code, the condition to start recording can test for value either bigger or smaller than the threshold.

The function invoked periodically to test the start trigger condition is m_rec_begin().

The function invoked periodically to test the stop trigger condition and perform the recording is m_recorder().

The recorder mode of operation is reported by the ASCII command TMODE (variable trace.Mode). The meaning of the codes stored is described in the table below:

TMODE codes	Modes of operation / status
0	Idle, stops recording if started
1	Recorder is armed – ready to start on beginning of motion
2	Recording in ongoing. Stop once the buffer is full.
3	Recording in circular buffer. Stop once the motion completes.

Once the data recording is completed, the host can use the command `PLAY` to get content of the recording buffers. The function implementing this request is `m_Play()`. It also formats binary packet response if the invocation context is packet command handler.

7.5 Motor Phasing

The Phasing procedure is intended to identify the absolute angle of the rotor of a brushless motor. This is needed by the control algorithms that calculate the orientation of the torque generating magnetic flux. The execution of the phasing procedure is required after each power cycle when incremental encoder feedback is used. The absolute encoders eliminate the need for such procedure. Still – they need at least once configuration of their phase offset – value that defines the rotor angle within the single turn encoder position. The selected phasing mode is defined by the value in the motor variable `phasing_mode`. The phasing algorithm to be used depends on different factors. The summary below describes the different phasing options as well as the pros and cons of each algorithm.

Phasing Procedure	Description
Forced Phasing <code>phasing_mode == 0</code>	<p>In this mode the firmware forms a voltage vector a known angle. It is formed by applying appropriate PWM duty cycle to each of the three phase outputs.</p> <p>The voltage is applied with magnitude defined by the motor variable <code>motor_power</code> for a duration defined in the motor variable <code>phasing_time</code>. These two variables have to be configured so that they will cause the motor to rotate its rotor such that it is oriented along the orientation of the magnetic flux. Once the time expires, the algorithm stores the current position and sets the phase origin 90degrees back from it.</p> <p>This procedure is implemented in the function <code>forced_phasing()</code> in the file <code>m_phasing.c</code></p> <p>The pros of this function are its simplicity and robustness. The cons are the small move in random direction the motor would make during the procedure execution. Another disadvantage is that the motor should have no static friction or gravity load that would affect the proper rotor orientation.</p>
Dithering Based Phasing <code>phasing_mode == 2</code>	<p>The dithering algorithm is derived from the Forced phasing algorithm – identifying the rotor position by observing its position after known flux is applied for a certain time.</p> <p>Unlike the Forced phasing algorithm, the Dithering algorithm does not wait for a certain time – instead it monitors the position change of the rotor. Once the motion direction is detected, the flux orientation is changed so that it cause change in the opposite direction. The magnitude of the flux angle changes is gradually reduced until the motion is no longer detected. The end result is motor phasing that only includes small motor vibrations for a short time as part of the phasing.</p> <p>This algorithm has the benefits of the Forced Phasing algorithm but without the drawback of unwanted motion. The cons are the need of carefully tuning the algorithm parameter in order to match the dynamic characteristics of the mechanical system the motor is attached to. The presence of static friction and gravity load are also undesired.</p> <p>The algorithm is implemented by the function <code>dither_phasing()</code> in the file <code>m_phasing.c</code></p>

7.6 Motor Homing

The homing procedure is required in applications that do not have absolute encoders or when the position of the absolute encoder is not calibrated. The execution of the home procedure executes series of moves and reads the feedback from external sensors to establish the origin of the coordinate system of the controlled axis.

The homing procedure can be executed in three different modes depending on the position feedback and home sensor available. They are described in the table below:

Homing Procedure	Description
Index Based	<p>This home procedure is started if the home_mask motor parameter is set to 0.</p> <p>This homing mode is used when a device uses a rotational axis and its coordinate system need to identify the angle where the orientation of the motor shaft is at 0 degree.</p> <p>The use of the index pulse as home position reference is only possible with incremental encoders with index pulse output. Normally it is labeled with Z+/Z-. Note that some encoders do not have differential index output. In this case the Z- input must be biased with 3.3V at the connector.</p> <p>The homing procedure starts a motion and arms the position capture hardware so that it would be triggered as soon as the index pulse is registered. Once the index position is captured, it is subtracted from the current position along with user defined home offset. This effectively adjusts the position offset and establishes the index position (plus the home offset) as new origin (zero) of the axis coordinate system.</p> <p>Once this is done, the axis target position is set to zero and a motion is started.</p>
Hard Stop Based	<p>This home procedure is executed when the home_mask motor parameter is set to 3.</p> <p>The hard stop based home procedure is polling the position error continuously. Once it exceeds 100 encoder counts, the current position is considered origin of the axis coordinate system.</p> <p>Similar to the Index based homing, user defined home offset is subtracted from the current position. Once this is done, the axis target position is set to zero and a motion is started.</p>
Home Sensor Based	<p>This homing procedure is executed when the home_mask is a non-zero value different than 3. The home_mask value is applied to logical AND operation with the digital inputs value. This effectively allows selection of the digital input to be considered "home switch".</p> <p>The execution of the home procedure starts with motion in direction dependent on the home switch status. The objective of the motion is to cause change in the input state. The triggering of the home switch is the indication of detecting vicinity of the coordinate system origin.</p> <p>Due to the lag in sensing the home sensor trigger (it is polled every 100 microseconds) the accuracy of the home sensor position is not sufficient in high-precision applications. For this reason, the homing continues with the Index Based procedure.</p>

The state diagram shows the states and the transitions between them. The execution flow is defined by the motor variable `home_mask` as described above.

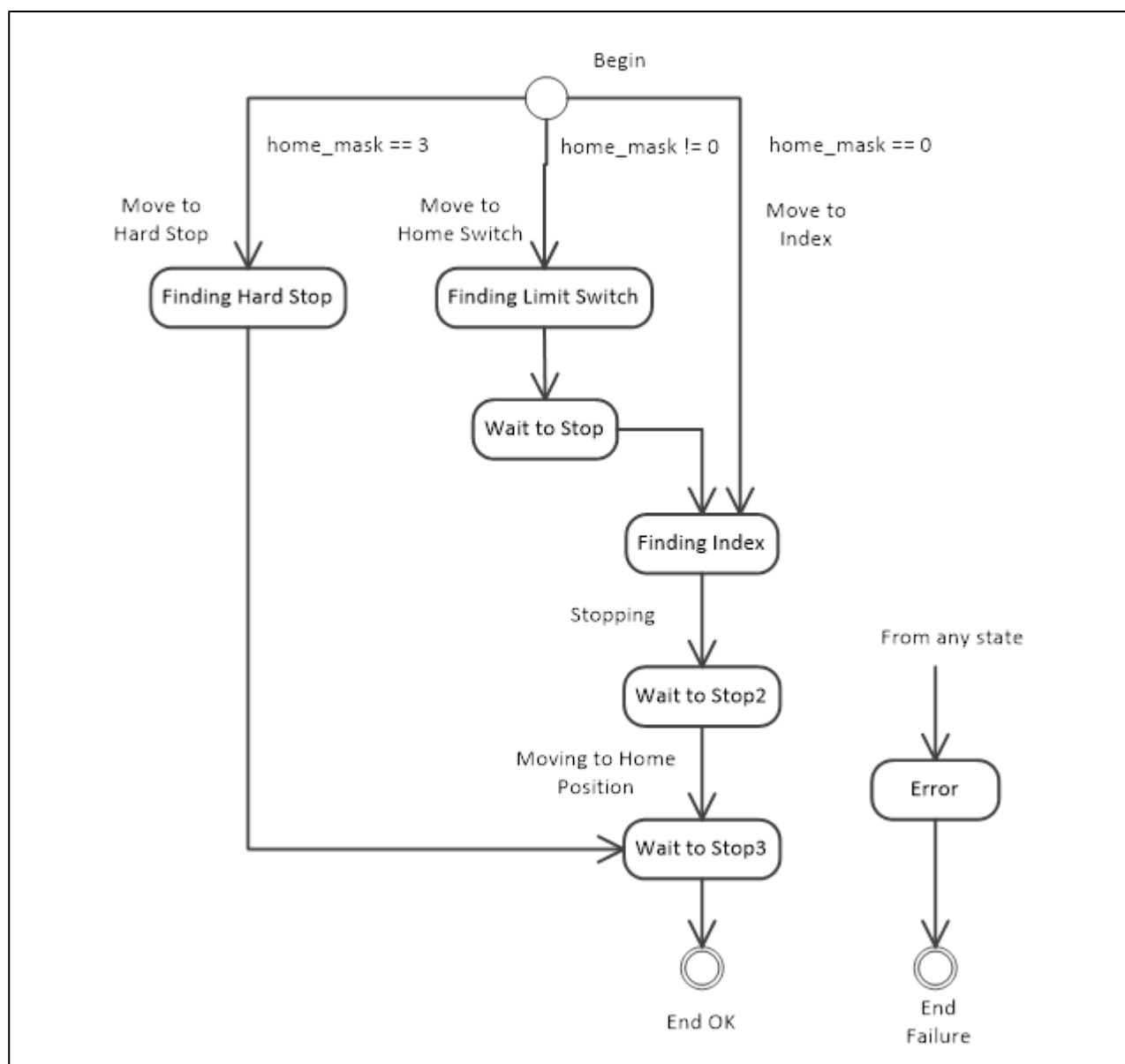


Figure 7.2 Homing Procedure State Machine

8. Host Communication

8.1 ASCII Communication Protocol

The ASCII protocol is based on commands consisting of ASCII characters terminated with Carriage Return (CR, ASCII 13). The controller responds with an optional data string followed by a prompt.

The ASCII protocols uses the following communication parameters:

115,200 bps, 8 data bits, 1 stop bit, no parity

When the command is accepted the prompt consist of CR, Line Feed (LF, ASCII 10) and Greater Than sign (>). When the command is rejected the prompt has Question Mark (?) instead of the Greater Than sign.

Example:

POS ; Host command terminated by CR

120 ; Reply Data String

> ; Reply Prompt

The variable names entered at the command prompt report the value of the referenced variable. If a parameter follows the name it is interpreted as a request to set the variable to a new value. Some variables are read-only. An attempt to set a value to them will be reported as invalid command. Examples:

>POS ; Request value of the variable POS

2100

>POS 2000 ; Set POS to a new value

>POS ; Report the new value

2000

>

The full set of ASCII commands is described in the Appendix A to this application note.

8.2 Binary Packet Communication Protocol

The Binary Packet Protocol is required when more than one motion controller is connected to the same serial interface using RS422 bus. In this case the address field of the packet allows the controllers to distinguish the packets and respond only to the ones addressed specifically to them. Another advantage of this protocol is the use of checksums that validate the integrity of the packet. This method increases the robustness of the communication and makes it especially valuable in noisy industrial environments.

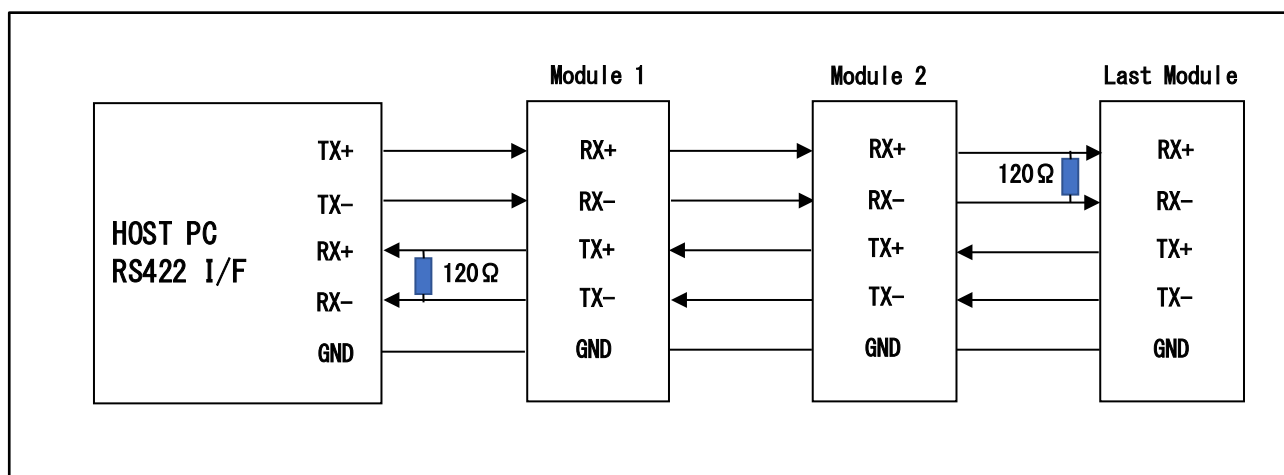
The Packet Protocol implemented in the half duplex. The bus topology uses one host and multiple slave modules like the Motor Solution Board or other compatible boards. The protocol is based on variable length packets for better bandwidth utilization.

The packets consist of fixed length header and variable length payload. The packet content is validated with 8-bit checksum. The reply length is dependent on the specific request and it too has variable length. Each reply starts with 2-byte status followed by optional data bytes. The reply completes with a checksum byte which is calculated as sum of all reply bytes.

The protocol allows addressing of up to 127 modules. Specific modules could be associated with a group address. After that they can be addressed at once by the host. A designated group master can optionally reply on behalf of the group. Finally, the protocol supports designated broadcasting address that can be used to address all modules regardless of their address or their association with a group.

8.2.1 Communication Bus

The Packet Protocol used with the Motor Solution Board is designed to achieve efficient communication between a host computer and multiple motion control modules connected on a RS422 bus. The figure below shows typical bus architecture:



Terminators of 120ohm are recommended at the receive line. They should be connected at the last module.

The communication data format uses 8 data bits, 1 stop bit, no parity control. The default communication baud rate upon startup is 115.2Kbps. It can be changed at run time, but after reset, it will go back to 115.2Kbps.

8.2.2 Modules Addressing

The communication between the host computer and the modules is half duplex. The host is always initiating the request and all modules listen. The one that recognizes its address responds to the host. Since the Motor Solution Board hardware and firmware is designed to drive two motors, a single module could respond to two different addresses – one for each channel.

The addresses assigned to each module are pre-defined and static. The factory-default setting assigns address 1 and 2 for the two channels on the Motor Solution Board. These addresses can be changed while the board is connected over RS232 and uses ASCII protocol. Once a new address is assigned, this firmware can save it along with the other controller parameters to the QSPI flash.

The protocol allows group addressing. The group address can be assigned at run time. One module could be designated as a group master. It responds on behalf of the group when the host addresses the whole group. This is useful for synchronization between multiple motion control modules.

There is a designated broadcast address 255 (0xFF) – useful for commands such as setting baud rate. All modules process the packet sent to this address, but none of them responds.

Address assignment procedure: use RS232 mode to set ADDR and GROUP variables. They can be saved to the QSPI flash with the SAVE command.

8.2.3 Request Packet Format

The generic packet format consists of a header and a payload. The header is fixed in length – it consists of 3 bytes. The payload length is variable. It depends on the context of the packet type which is always the first byte of the payload.

The request packet size is explicitly specified. The length of the payload is defined by the third byte of the header. The reply packet size is implicitly defined. It depends on the specific request.

Generic Request Packet:

Header			Payload		
Start Byte (0xAA)	Address Byte (node ID)	Length Byte (1 – 255)	Pkt Code Byte (see below)	Variable length (0..254)	Check Sum Byte
0	1	2	3	N bytes	4 + N

Checksum calculation includes all bytes except the first one (0xAA).

Generic Reply Packet:

Reply Code	Variable length	Check Sum
X	N	X + N

Checksum Calculation for the reply includes all bytes. Request Packet Checksum error is indicated by reply code 255.

Request Packet Type Codes:

Packet Type	Command Description	Req Packet Byte Size	Reply Packet Byte Size
0	Get Module Data	5	7
1	Data Report Request	7	Variable
2	Function Call	6	3
3	Initialize PVT Mode	6	3
4	Stream Position/Velocity set points	5 + points * 8	3
5	Set Parameter (16-bit or 32-bit)	8 or 10	3
6	Get Parameter	6	5 or 7
7	Get Trace Buffer	6	Variable
15	Set Communication Baud Rate	6	3

Reply codes:

Reply Code	Description
0	Command Accepted
1	Invalid Packet Code
2	Invalid Parameter Index
3	Invalid Data Size / Format
4	Parameter Value Out of Range
5	Invalid Request - Attempt to set a Read-Only variable
6	Interlock Active
7	Controller Fault
8	Invalid State
9	Buffer Overflow
10	Execution Error
255	Checksum error

The detailed description of the different packet formats is included as an appendix of this manual.

8.3 EtherCAT Support



8.3.1 Software Configuration

Software configuration diagram containing the protocol stack is shown the following.

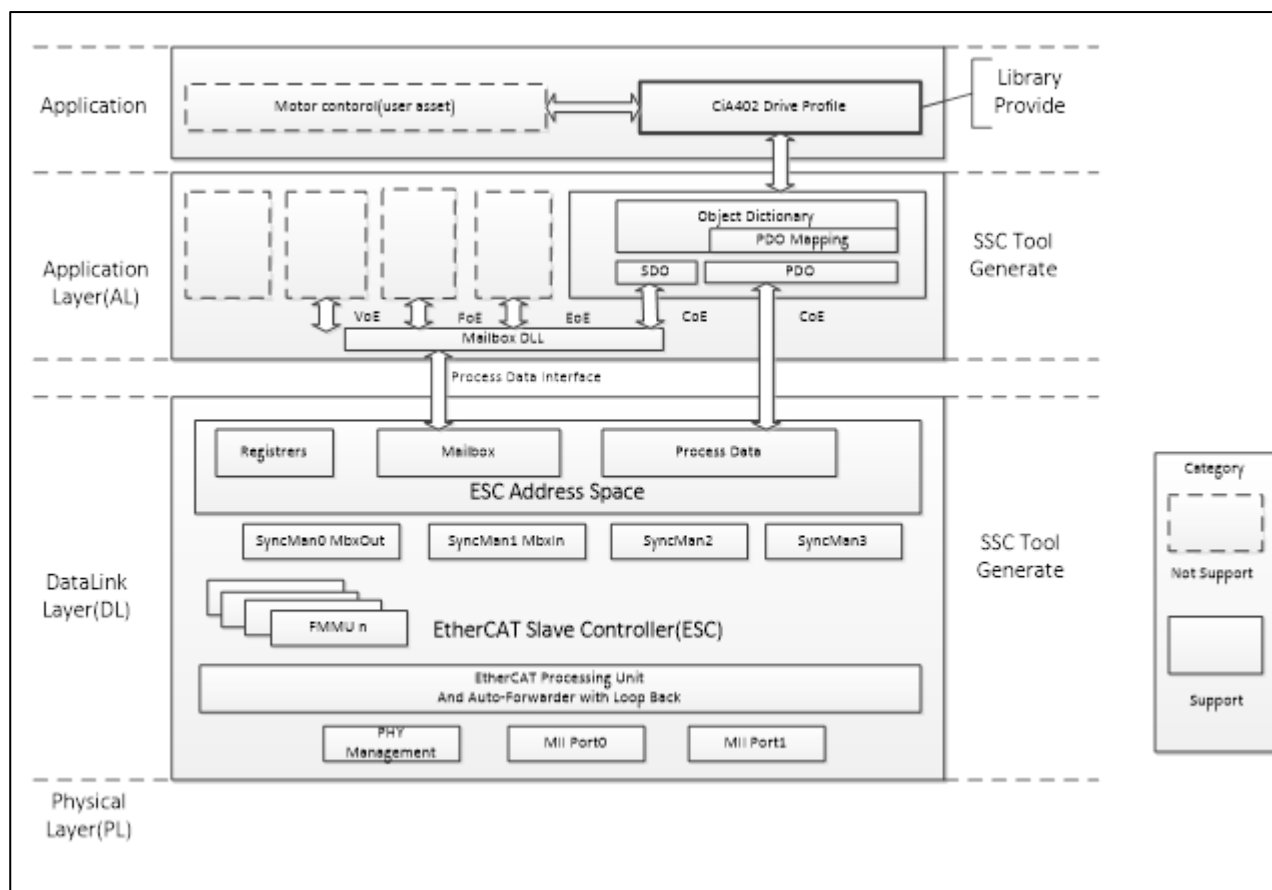


Figure 8.1 Software Configuration Diagram

SSCTool: Slave Stack Code Tool produced by Beckhoff Automation GmbH

ESC: EtherCAT Slave Controller

The CAN application protocol over EtherCAT (CoE) is used as the AL layer to adapt the EtherCAT communication in the form of CiA402 drive profile as shown in Figure 8.1. The process data objects (PDO) were used at the time of real-time data transfer by cyclic communication. PDO has two elements, one is RxPDO (receiving data from the PLC) and the other is TxPDO (sending status information and so to the PLC). In the case of asynchronous communication, the object dictionary is read and written by using the mailbox communication (SDO).

8.3.2 CiA402 Drive Profile

CiA402 drive profile is the device profile for drivers and motion controllers and mainly defines functional operations for servo drives, sine wave inverter and stepping motor controller. In this profile, the multiple operating modes and corresponding parameters are defined as an object dictionary. Also, Finite State Automaton: FSA to define the internal and external behavior in every state is included. To change the status, set the controlword object, then statusword which shows the current status reflects the result after transition. The controlword and various command value (like target position) are assigned to RxPDO, and the statusword and various current condition (like actual position) are assigned to TxPDO. Please see the contents of the CiA402 standard for more details.

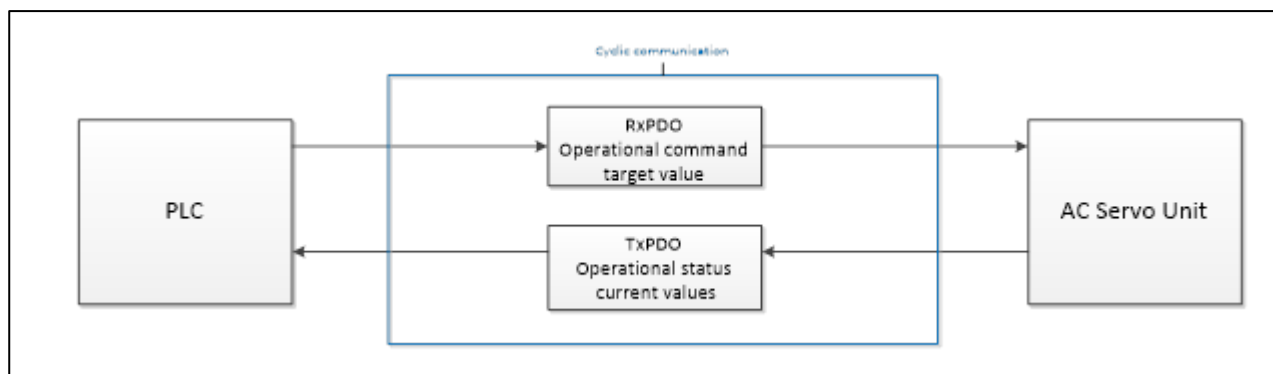


Figure 8.2 CiA402 communication flow

Written Standards of CiA402 :

IEC 61800-7-201 Edition 1.0

Adjustable speed electrical power drive systems Part 7-201:

Generic interface and use of profiles for power drive systems Profile type 1 specification

IEC 61800-7-301 Edition 1.0

Adjustable speed electrical power drive systems Part 7-301:

Generic interface and use of profiles for power drive systems Mapping of profile type 1 to network technologies

8.3.2.1 Operation Mode

In this application note, the following modes are supported from among the operation mode defined in CiA402 standard.

Table 8.1 Supported Operation Mode List

Operation Mode	Support
Profile position mode	Yes
Velocity mode (frequency converter)	No
Profile velocity mode	No
Profile torque mode	No
Homing mode	No
Interpolated position mode	No
Cyclic synchronous position mode	Yes
Cyclic synchronous velocity mode	Yes
Cyclic synchronous torque mode	No
Cyclic synchronous torque mode with commutation angle	No
Manufacturer specific mode	No

8.3.2.2 State Transition

FSA defined in CiA402 standard is supported in this application note.

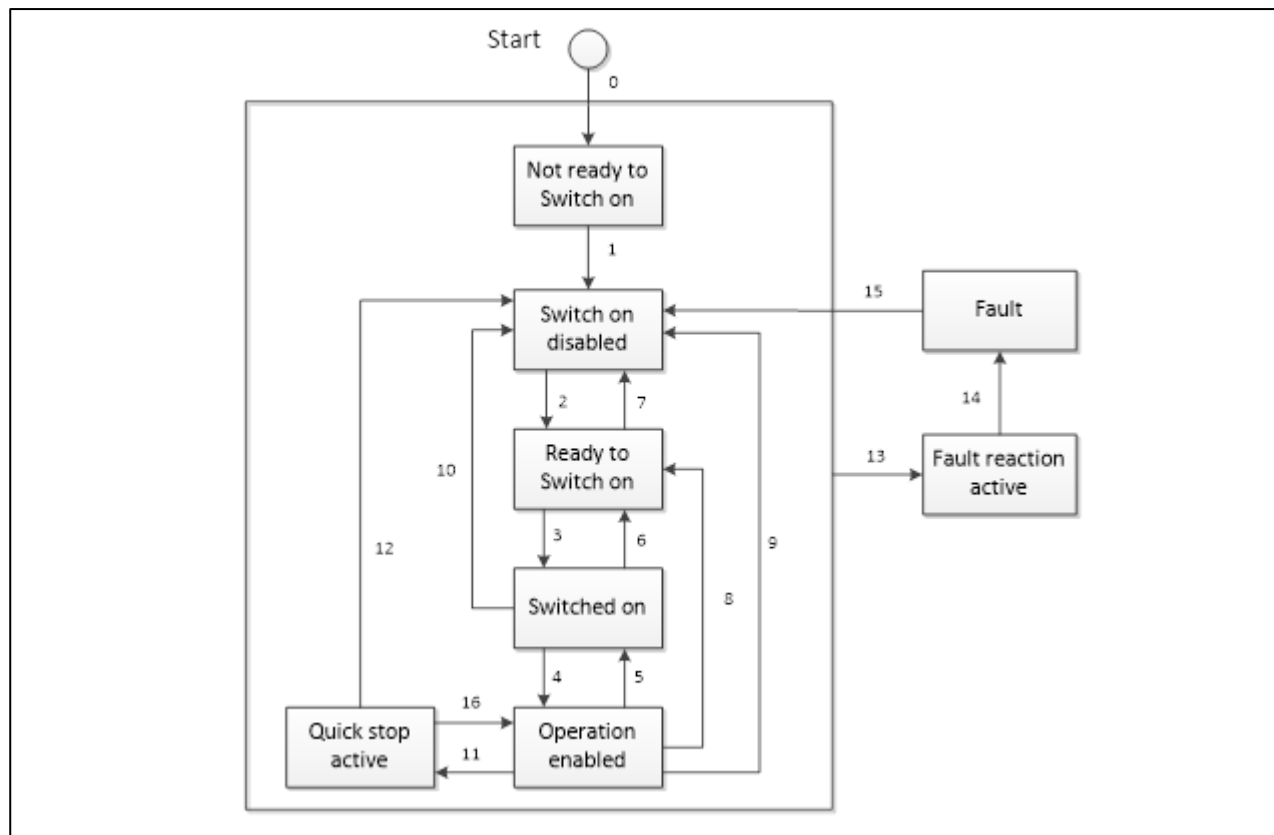


Figure 8.3 CiA402 State Transition Diagram

8.3.2.3 Object Dictionary

The following table shows the object dictionaries supported in this document.

Table 8.2 Support Object Dictionary List

Operation Mode	OBJECT Name	INDEX	Category	Access	Data Type	PDO Mapping
Profile position mode	Target position	0x607A	Mandatory	rw	INT32	RxPDO
	Profile velocity	0x6081	Mandatory	rw	UINT32	RxPDO
	Profile acceleration	0x6083	Mandatory	rw	UINT32	RxPDO
	Profile deceleration	0x6084	Optional	rw	UINT32	RxPDO
	Motion profile type	0x6086	Optional	rw	INT16	RxPDO
	Profile jerk	0x60A4	Optional	rw	UINT32	RxPDO
Cyclic synchronous position mode	Torque actual value	0x6077	Recommended	ro	INT16	TxPDO
	Target position	0x607A	Mandatory	rw	INT32	RxPDO
	Motion profile type	0x6086	Optional	rw	INT16	RxPDO
	Profile jerk	0x60A4	Optional	rw	UINT32	RxPDO
	Velocity offset	0x60B1	Optional	rw	INT32	RxPDO
	Torque offset	0x60B2	Recommended	rw	INT16	RxPDO
Cyclic synchronous velocity mode	Torque actual value	0x6077	Recommended	ro	INT16	TxPDO
	Torque offset	0x60B2	Recommended	rw	INT16	RxPDO
	Target velocity	0x60FF	Mandatory	rw	INT32	RxPDO
All	Error code	0x603F	Optional	ro	UINT16	TxPDO
	Controlword	0x6040	Mandatory	rw	UINT16	RxPDO
	Statusword	0x6041	Mandatory	ro	UINT16	TxPDO
	Modes of operation	0x6060	Optional	rw	INT8	SDO
	Modes of operation display	0x6061	Optional	ro	INT8	TxPDO
	Position actual value	0x6064	Mandatory	ro	INT32	TxPDO
	Velocity actual value	0x606C	Conditional	ro	INT32	TxPDO

8.3.2.4 Implement Motor Control Program

According to the table below and the CiA402 standard, implement the motor application. Each function links the number of each state transition in the figure of "CiA402 State Transition Diagram" and the corresponding function is called in case of state transition. Describe the motor control program or call operation of the main CPU in each function.

Table 8.3 CiA402 Protocol Stack I/F Function List

CiA402_StateTransition1	
	<u>Description</u> This function is used when the state transition 1 occurred Describe the operation in the case of the state transition.
	<u>Usage</u> #include "cia402appl.h"
	<u>Parameters</u> TCiA402Axis *pCiA402Axis
	<u>Return Value</u> 0 Normal end 1 Error
	<u>Remark</u> <u>Remark</u> In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition2	
	<u>Description</u> This function is used when the state transition 2 occurred Describe the operation in the case of the state transition.
	<u>Usage</u> #include "cia402appl.h"
	<u>Parameters</u> TCiA402Axis *pCiA402Axis
	<u>Return Value</u> 0 Normal end 1 Error
	<u>Remark</u> In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.

CiA402_StateTransition3	
<u>Description</u>	This function is used when the state transition 3 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition4	
<u>Description</u>	This function is used when the state transition 4 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition5	
<u>Description</u>	This function is used when the state transition 5 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.

CiA402_StateTransition6	
<u>Description</u>	This function is used when the state transition 6 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition7	
<u>Description</u>	This function is used when the state transition 7 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition8	
<u>Description</u>	This function is used when the state transition 8 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.

CiA402_StateTransition9	
<u>Description</u>	This function is used when the state transition 9 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition10	
<u>Description</u>	This function is used when the state transition 10 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition11	
<u>Description</u>	This function is used when the state transition 11 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.

CiA402_StateTransition12	
<u>Description</u>	This function is used when the state transition 12 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_LocalError	
<u>Description</u>	This function is used when the state transition 13 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	UINT16 ErrorCode
<u>Return Value</u>	none
<u>Remark</u>	If the error corresponding to the state transition 13 occurs, call this function after processing required and saving data at error location
CiA402_StateTransition14	
<u>Description</u>	This function is used when the state transition 14 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.

CiA402_StateTransition15	
<u>Description</u>	This function is used when the state transition 15 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
CiA402_StateTransition16	
<u>Description</u>	This function is used when the state transition 16 occurred Describe the operation in the case of the state transition.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	In the case of error occurrence during processing, exit the function by setting the appropriate values for each object in accordance with CiA402 standard . If 1 is set to return value, state transition does not occur.
APPL_MOTOR_MotionControl_Main	
<u>Description</u>	Implement the motion control code when CiA402 FSA state is Operation enabled. Describe the process for each mode of operation.
<u>Usage</u>	#include "cia402appl.h"
<u>Parameters</u>	TCiA402Axis *pCiA402Axis
<u>Return Value</u>	0 Normal end 1 Error
<u>Remark</u>	

8.3.2.5 Special Instruction for implement protocol stack

Note1*: Alignment of a structure at mailbox use.

CiA402 protocol stack uses not only PDO but SDO communication. Please use “pack” option and others to the structure for SDO communication in order to ensure the alignment.

Note2*: Watchdog timer for EtherCAT

When TwinCAT + PC is used as master device, watchdog timer error may happen because of the lack of real-time quality for the cyclic SM event. In this application note, the sample code is created by the SSC tool at invalid setting of watchdog timer and checked the operation.

8.3.3 Shared Memory

8.3.3.1 Operation Outline

Contention for access between the applications may occur in the situations where user applications for the CPU0 and CPU1 share data. This driver realizes exclusive access control by using the semaphore registers to monitor the availability of resources in the form of shared memory.

For monitoring, the driver reads the semaphore registers. The semaphore bit in each semaphore register indicates whether or not the target resource is available. With the read-and-clear function enabled, if the target resource is available when the corresponding register is read, the semaphore bit in the relative register is cleared to 0 to indicate that the resource is in use.

Table 8.4 Semaphore Register n (SYTSEMF_n where n = 0 to 7)

Symbol	Symbol	Bit Name	Function	R/W
b0	SEMF _n	Semaphore	0: Resource is in use 1: Resource is available	R/W ¹
b31-b1	Reserved	Reserved	———	R/W

Note 1. Writing 1 to this bit retains its value. Reading this bit while the read-and-clear function is enabled clears its value to 0.

8.3.3.2 Software Operation

The following describes the procedure of semaphore control by the shared memory driver using the semaphore registers.

1. A core reads the semaphore register for the resource it requires. If the value of the semaphore bit is 0, the core continues to read the register until the value of the semaphore bit becomes 1.
2. Reading the semaphore bit as 1 confirms that the target resource is available. At this time, the bit is cleared to 0, indicating that the resource is in use by the given core.
3. Access a shared memory.
4. When the access is completed, write 1 to the relative bit to indicate that the resource is available.
5. In the cases of the access at points marked (1), (2), and (3) in Figure 8.4, the resource is available and access to the shared memory proceeds immediately. In the case of the attempted access at points marked (4), the access at the corresponding previous point (3) is still in progress. The core seeking access at point (4) waits until access preceding from the previous point (3) is completed and the resource becomes available. Contention for access is thus resolved.

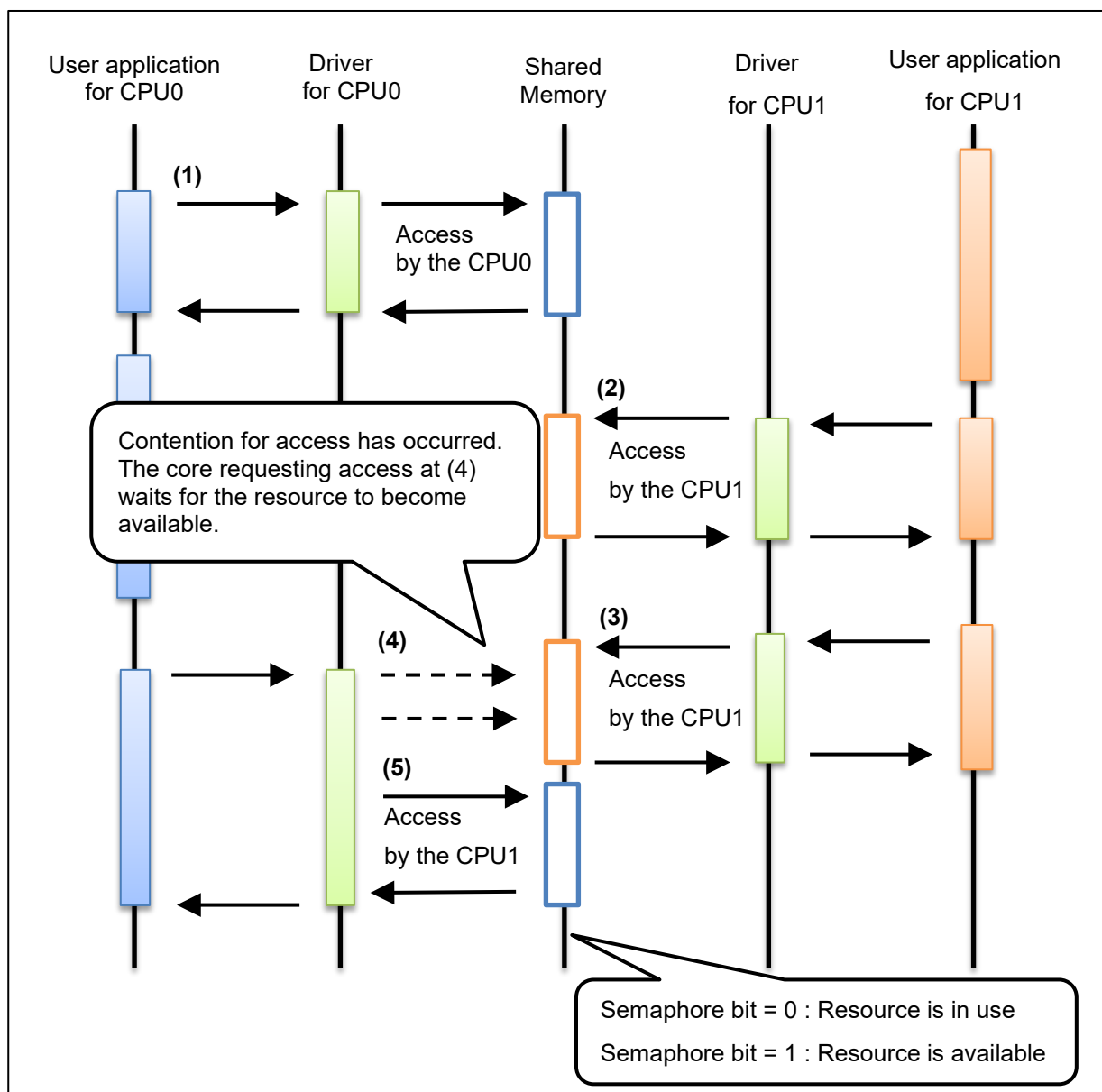


Figure 8.4 Software Operation Sequence

8.3.3.3 Software Block Diagram

The system block diagram for the shared memory driver is shown in the figure below. The areas in the red broken-line frames are the operations covered by the shared memory driver.

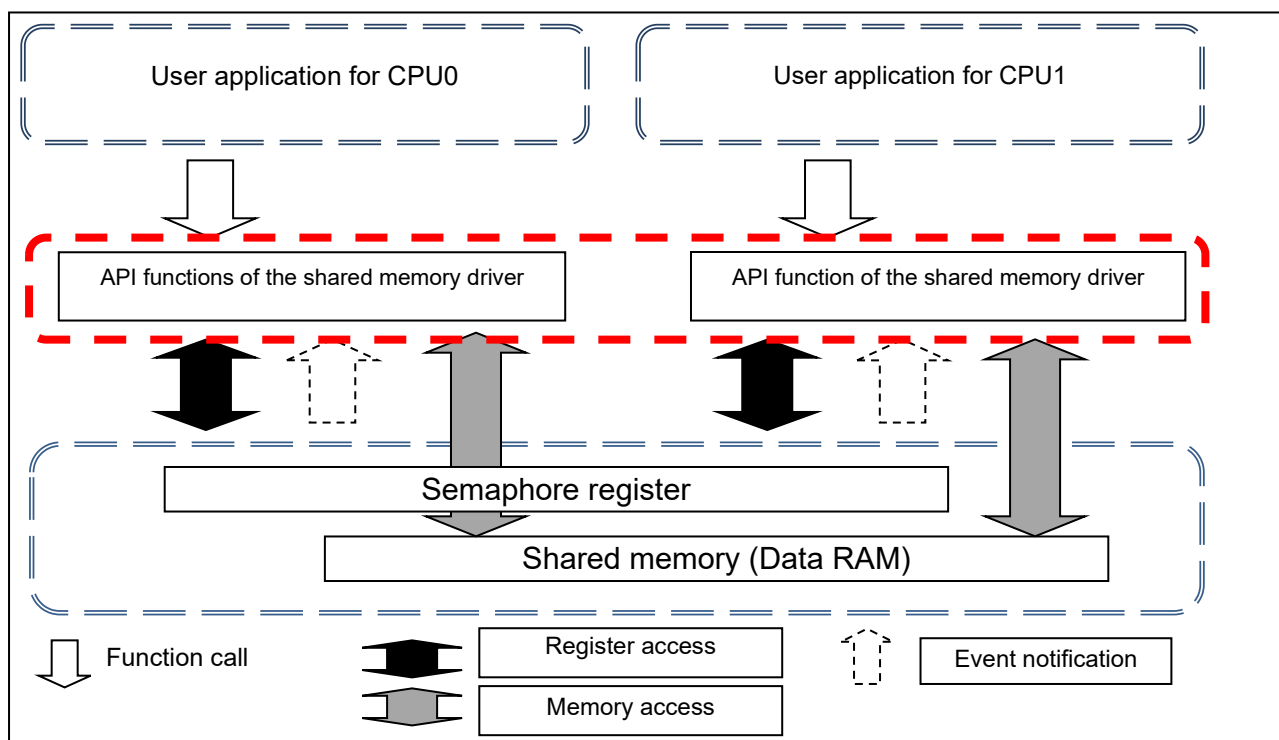


Figure 8.5 Shared Memory Driver System

8.3.3.4 Software State Transitions

Transition of the states of the shared memory driver are shown in the figure below.

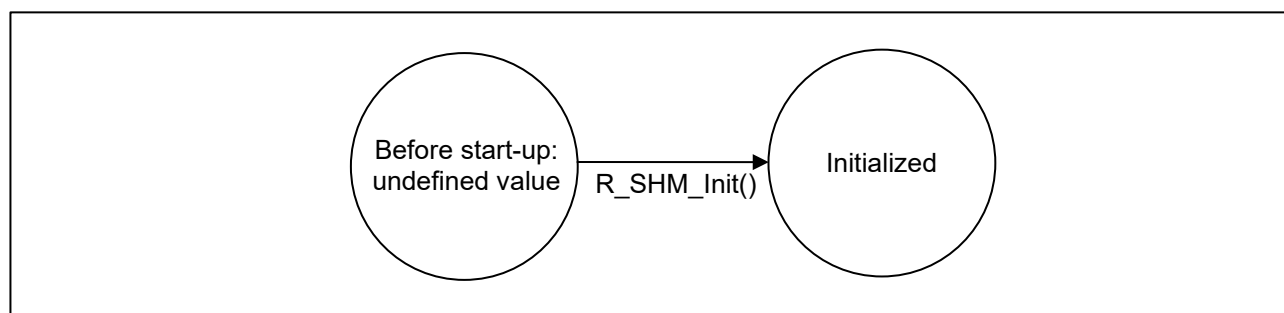


Figure 8.6 State Transition of the Shared Memory Driver

8.3.3.5 Required Memory Size

The area of shared memory is shown in the figure below. To this driver, the first 4KB of data RAM are reserved as shared memory.

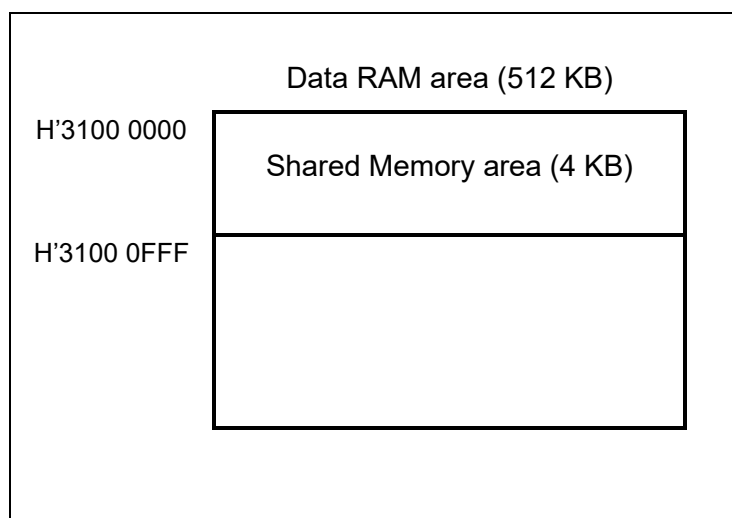


Figure 8.7 Shared Memory Area

8.3.3.6 Primitive Data Types

The primitive data types used in the shared memory driver are listed in the table below.

Table 8.5 Primitive Data Types

Symbol	Description
int8_t	8-bit signed integer (defined in the standard library)
int16_t	16-bit signed integer (defined in the standard library)
int32_t	32-bit signed integer (defined in the standard library)
int64_t	64-bit signed integer (defined in the standard library)
uint8_t	8-bit unsigned integer (defined in the standard library)
uint16_t	16-bit unsigned integer (defined in the standard library)
uint32_t	32-bit unsigned integer (defined in the standard library)
uint64_t	64-bit unsigned integer (defined in the standard library)

8.3.3.7 Definition of Data Structure

The table below lists the data structures.

Table 8.6 Data Structures

Definition of Structures	Description	Definition File
st_shm structure	The shared memory is stored in this structure.	r_shm_config.h

8.3.3.8 Global Variable

The table below lists the global variable.

Table 8.7 Global Variable

Variable	Variable	Description
st_shm structure	SHM	The shared memory is stored in this variable (structure).

8.3.3.9 Definitions of Values (Macros)

The table below lists the macros.

Table 8.8 Constant Macros

Macro Name	Value	Description	Definition File
SHM_SEMFNO_0	0	Semaphore register 0	r_shm.h
SHM_SEMFNO_1	1	Semaphore register 1	r_shm.h
SHM_SEMFNO_2	2	Semaphore register 2	r_shm.h
SHM_SEMFNO_3	3	Semaphore register 3	r_shm.h
SHM_SEMFNO_4	4	Semaphore register 4	r_shm.h
SHM_SEMFNO_5	5	Semaphore register 5	r_shm.h
SHM_SEMFNO_6	6	Semaphore register 6	r_shm.h
SHM_SEMFNO_7	7	Semaphore register 7	r_shm.h
SHM_SEMFNO_TOTAL	8	The highest number of the semaphore register to be specified	r_shm.h
SHM_SYS_SEMFNO	0	User-defined semaphore register number The semaphore register number is specified with this constant.	r_shm.h

8.3.3.10 Definition of Values (Enumeration Types)

The shared memory driver does not use enumeration types.

8.3.3.11 Error Codes

In the shared memory driver, 0 is returned for normal termination and a non-zero value is returned for errors. The table below lists the error codes.

Table 8.9 Error Codes

Macro Name	Value	Meaning	Definition File
SHM_SUCCESS	0	Normal termination	r_shm.h
SHM_ERR	-1	Error termination	r_shm.h

8.3.3.12 Functions

The interface functions of the shared memory drivers for the connected drivers are listed in the table below.

Table 8.10 API Functions for Shared Memory Driver

Function Name	Meaning	Definition File
R_SHM_Init	Initialize the shared memory driver	r_shm.h
R_SHM_memcpy	Load n bytes of data from the shared memory	r_shm.h
R_SHM_Load_uint32	Load four bytes of unsigned int type data from the shared memory	r_shm.h
R_SHM_Load_int32	Load four bytes of signed int type data from the shared memory	r_shm.h
R_SHM_Load_uint16	Load two bytes of unsigned int type data from the shared memory	r_shm.h
R_SHM_Load_int16	Load two bytes of signed int type data from the shared memory	r_shm.h
R_SHM_Load_uint8	Load one byte of unsigned int type data from the shared memory	r_shm.h
R_SHM_Load_int8	Load one byte of signed int type data from the shared memory	r_shm.h
R_SHM_Set_uint32	Set uint 4 bytes data to shared memory address.	r_shm.h
R_SHM_Set_int32	Set int 4 bytes data to shared memory address.	r_shm.h
R_SHM_Set_uint16	Set uint 2 bytes data to shared memory address.	r_shm.h
R_SHM_Set_int16	Set int 2 bytes data to shared memory address.	r_shm.h
R_SHM_Set_uint8	Set uint 1 byte data to shared memory address.	r_shm.h
R_SHM_Set_int8	Set int 1 byte data to shared memory address.	r_shm.h
R_SHM_Get_uint32	Read uint 4 bytes data from shared memory address.	r_shm.h
R_SHM_Get_int32	Read int 4 bytes data from shared memory address.	r_shm.h
R_SHM_Get_uint16	Read uint 2 bytes data from shared memory address.	r_shm.h
R_SHM_Get_int16	Read int 2 bytes data from shared memory address.	r_shm.h
R_SHM_Get_uint8	Read uint 1 byte data from shared memory address.	r_shm.h
R_SHM_Get_int8	Read int 1 byte data from shared memory address.	r_shm.h

8.3.3.13 Structures Defined by Shared Memory Driver

The table below lists the structures defined for the shared memory driver and used by the user program.

Table 8.11 st_shm structure

Member Type and Name	Range of Normal Values	Usage and Remark
Any members	Any values	Members of the structures need to be defined to suit the application.

8.3.3.14 Flowchart

(1) Initialization Processing of Driver

The following flowchart shows the initialization processing of a driver.

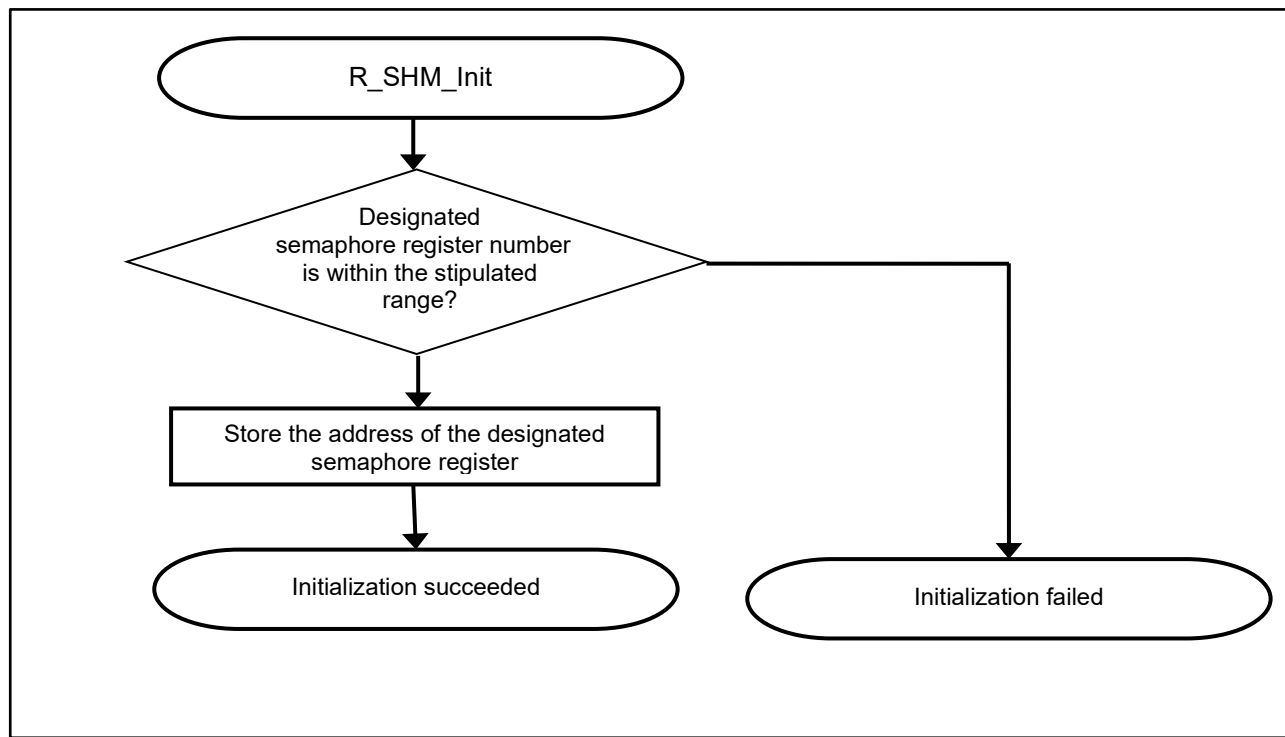


Table 8.12 Driver Initialization Processing

(2) Processing to Load n Bytes of Data from the Shared Memory

The following flowchart shows the processing to load n bytes of data from the shared memory.

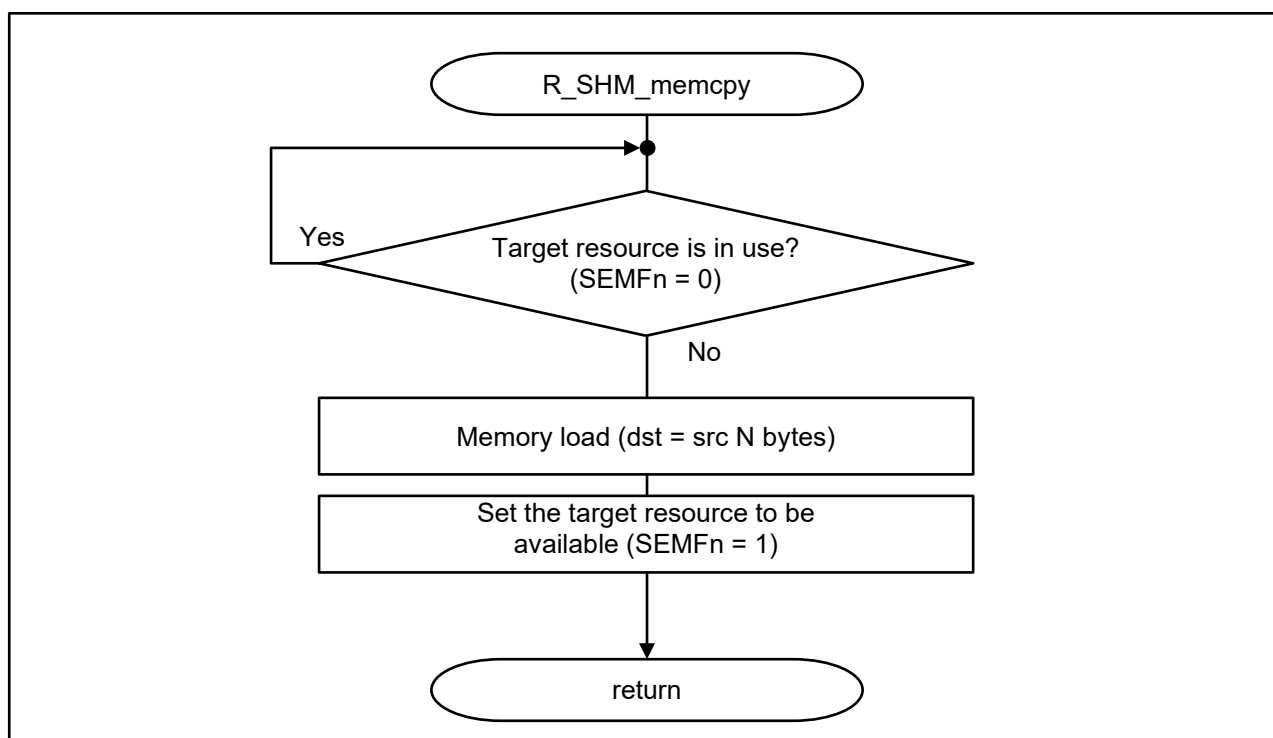


Figure 8.8 Processing to Load n-Bytes of Data from the Shared Memory

The flowcharts are the same for other processing to load data (`R_SHM_Load_uint32`, etc.).

(3) Processing to Set n Bytes of Data to the Shared Memory Address

The following flowchart shows the processing to set n bytes of data to the shared memory address.

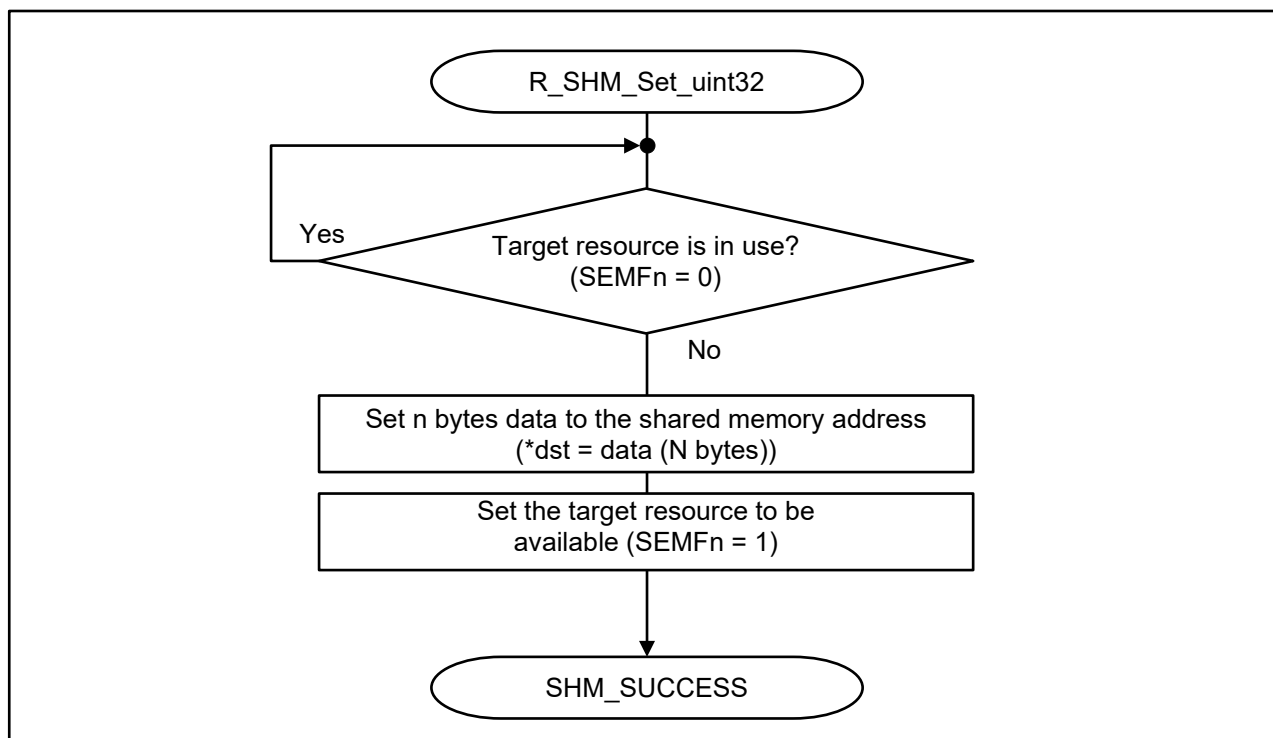


Figure 8.9 Processing to Set n-Bytes of Data to the Shared Memory Address

N= signed 4 bytes, unsigned 4 bytes, signed 2 bytes, unsigned 2 bytes, signed 1 byte, unsigned 1 byte

(4) Processing to Read n Bytes of Data from the Shared Memory Address

The following flowchart shows the processing to read n bytes of data from the shared memory address.

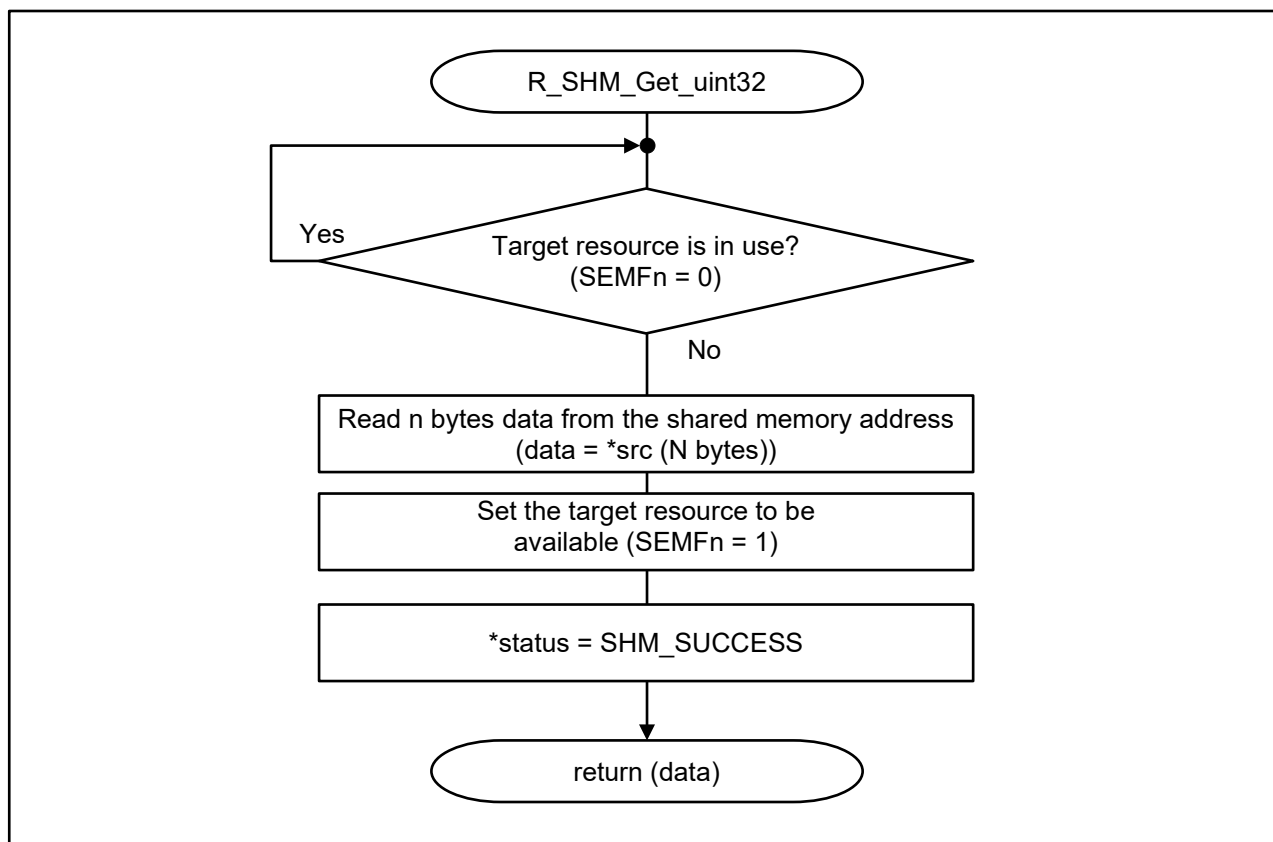


Figure 8.10 Processing to Read n-Bytes of Data from the Shared Memory Address

N= signed 4 bytes, unsigned 4 bytes, signed 2 bytes, unsigned 2 bytes, signed 1 byte, unsigned 1 byte

8.3.3.15 Usage Example

Before creating a user application, define the semaphore register numbers to be used in the shared memory driver and the members of the shared memory structures.

- Use the header file (r_shm_config.h) to define the driver structures.
- Use the same header file to define the driver structures for the CPU0 and CPU1.
- The location of the header file is described in Section “File Structure”.

(1) Defining the Semaphore Register Numbers

The semaphore register numbers to be used in the shared memory driver are designated between 0 and 7.

```
#define SHM_SYS_SEMFNO    (0)    // System SEMFn for shared memory access (n = 0-7)
```

Figure 8.11 Example of selecting a Semaphore Register

(2) Defining the Structure Members in the Shared Memory

The members in the shared memory are defined according to the user application. Be sure that the defined members do not exceed the size of the shared memory.

```
/* ---- Shared memory struct ---- */  
struct st_shm  
{  
    TAxis  Axis[2];  
};
```

Figure 8.12 Example of defining a Structure Member in the Shared Memory

9. Related Documents

RZ/T2M Motor Solution Board Hardware Manual (R01AN5986EJ)

RZ/T2M Motor Solution Kit Startup Manual (RZ/T2M Motion Control Utility) (R01AN5989EJ)

RZ/T2M Motor Solution Kit Startup Manual (EtherCAT) (R01AN6470EJ)

RZ/T2M Group User's Manual Hardware (R01UH0916EJ)

Appendix A ASCII Communication Protocol Commands

The table below represents unified enumeration of all configuration parameters, status variables and functions in this firmware.

R/O = Read Only, R/W = Read / Write, Cmd – Command

Par ID	Name	Byte Size	Access	Description
0	STA	4	R/O	Status word (reported as 4 digit Hex word). The bit flags are documented below.
1	ERR	2	R/O	Position error – the difference between the actual and the expected encoder position. The value is meaningful only when servo control is on.
2	ADC1	2	R/O	Current reading as reported by the ADC channel 1. The correspondence between the number reported and the actual current is hardware dependent.
3	ADC2	2	R/O	Current reading as reported by the ADC channel 2
4	TC	2	R/O	Total current – calculated as a sum of the modulus of ADC1 and ADC2 currents
5	CV	2	R/O	Current velocity as encoder counts per sample interval (the position loop cycle time)
6	PVT	2	R/O	Number of points in the PVT buffer
7	ITIME	2	R/W	Time interval between consecutive PVT points
8	VEL	4	R/W	Command Velocity – (encoder counts per sample interval) * 65536
9	ACC		R/W	Command Acceleration – (encoder counts per sample interval squared) * 65536
10	DEC	4	R/W	Command Deceleration – (encoder counts per sample interval squared) * 65536. If the value is zero then the Acceleration value is used in its place.
11	AJERK	4	R/W	Acceleration Jerk (0 – 1000)
12	DJERK	4	R/W	Deceleration Jerk (0 – 1000)
13	PRO	2	R/W	Velocity profile mode
14	KP	2	R/W	Proportional Gain in the position control loop algorithm (0 – 32767)
15	KI	2	R/W	Integral Gain in the position control loop algorithm (0 – 32767)
16	KD	2	R/W	Differential Gain in the position control loop algorithm (0 – 32767)
17	IL	2	R/W	Integral Limit in the position control loop algorithm (0 – 32767)
18	VFF	2	R/W	Velocity Feed Forward in the position control loop algorithm (0 – 32767)
19	AFF	2	R/W	Acceleration Feed Forward in the position control loop algorithm (0 – 32767)
20	MAX	2	R/W	Maximum position error (0 – 32767)
21	ETIME	2	R/W	Maximum position error time (in sample intervals)
22	DS	2	R/W	Position Loop Derivative Sample Interval (sample interval) in multiples of 50us. Setting of 2 indicates 100 us sample interval.

Par ID	Name	Byte Size	Access	Description
23	MLIMIT	2	R/W	Motor output limit from the position loop PID regulator (0 – 32767)
24	BIAS	4	R/W	Value to be added to the output of the PID regulator continuously
25	ASTOP	2	R/W	Automatic stop mode. Determines the action after position error is exceeded: 0 – stop the motion, 1 – stop the servo control in addition to stopping the motion.
26	PIMODE	2	R/W	Phase initialization mode: 0 – Forced, 1 – Hall Based, 2 – Dithering Based
27	PITIME	2	R/W	Phase initialization time [sample intervals]
28	PIOUT	2	R/W	Phase initialization output (32767 = 100%)
29	PMAP	2	R/W	Phase mapping to PWM output channels. Allows software defined wiring between the controller outputs and the motor windings. Values should range from 0 to 5.
30	PORIGIN	4	R/W	Phase origin – the encoder position within the current motor revolution where the flux is at 0 degree.
31	PCMODE	2	R/W	Phase commutation mode: 0 = voltage controlled Space Vector Modulation, 1 = Field Oriented Control, 2 = Hall based commutation, 3 = Host defined phase angle,
32	PVECTOR	4	R/W	Phase vector orientation times 60 degree. Values should range from 0 to 5.
33	PPAIRS	2	R/W	Pole pairs – this parameter reflects motor rotor construction.
34	PCOUNTS	4	R/W	Encoder counts per electrical cycle (encoder counts per rev divided by the number of pole pairs)
35	ECPR	4	R/W	Encoder counts per revolution – encoder resolution
36	PIOFFS	4	R/W	Phase Initialization Offset – position offset added at the end of the phase initialization procedure
37	PANGLE	4	R/O	The current angle of the magnetic flux
38	PADV	4	R/W	Phase advance gain (not used)
39	VCOMP	4	R/W	Velocity compensation (not used)
40	CLIMIT	2	R/W	Current limit threshold
41	CTIME	2	R/W	Current limit time (in sample intervals)
42	IDM	4	R/O	Direct (heat generating) current
43	IQM	4	R/O	Quadrature (torque generating) current
44	IQERR	4	R/O	Quadrature current error
45	QKP	2	R/W	Proportional gain in the Quadrature current control loop
46	QKI	2	R/W	Integral gain in the Quadrature current control loop
47	PCT	2	R/W	Position capture mode (only index capture supported)
48	HMASK	2	R/W	Home switch mask – defines the type of the home algorithm and the specific input wired to home sensor
49	INVERT	2	R/W	Home switch invert mask – defines the inversion of the home switch input so that it will be searched in the correct direction.
50	HINVERT	4	R/W	Hall sensor signal inversion: 0 – no inversion, 1 – inverted

Par ID	Name	Byte Size	Access	Description
51	HSIFT	4	R/W	Hall sensor position shift (not used)
52	HPOS	4	R/W	Hall sensor position change – the last position where the hall sensors changed their status
53	EMASK	2	R/W	Digital inputs error mask – defines digital input as an interlock that can trigger motion stop if triggered.
54	ECP	4	R/W	Command position where the following error exceeded the maximum threshold
55	ECV	4	R/W	Velocity at which the following error exceeded the maximum threshold
56	EPO	4	R/W	Actual position where the following error exceeded the maximum threshold
57	U	4	R/W	The output voltage of phase U. (32767 = 100% duty cycle)
58	V	4	R/W	The output voltage of phase V. (32767 = 100% duty cycle)
59	W	4	R/W	The output voltage of phase W. (32767 = 100% duty cycle)
60	TYPE	2	R/W	Module type: 1 = single channel, 2 = dual channel with electronic gearing, 3 = dual channel / two independent motors
61	HTYPE	2	R/W	Hall sensors type (only parallel type supported)
62	HOFFS	4	R/W	Home offset – value added to the zero coordinate at the end of the home procedure.
63	DATA0	4	R/W	User defined data – no specific use in the firmware
64	DATA1	4	R/W	
65	DATA2	4	R/W	
66	DATA3	4	R/W	
67	DATA4	4	R/W	
68	DATA5	4	R/W	
69	DATA6	4	R/W	
70	DATA7	4	R/W	
71	PHASES	2	R/W	Defines the motor type: 2 – DC Brush type, 3 – Brushless DC / PMSM motor type
72	BRAKE	2	R/W	Brake control mode (not used)
73	TMODE	2	R/W	Trace mode: 0 - Idle, 1 = Armed – waiting for trigger event, 2 = Start now, Stop on buffer full, 3 = Start now, Stop on end of motion
74	TRATE	2	R/W	Data recorder rate (in 50us intervals)
75	TLEVEL	4	R/W	Data recorder threshold level
76	SIM	2	R/W	Enables (1) or disables(0 – default) encoder simulation
77	TIMER	2	R/W	Timer register that generates the PWM carrier frequency.
78	ADDP	4	R/W	Add Position set point for PVT streaming
79	ADDV	4	R/W	Add Velocity set point for PVT streaming
80	ABS	4	R/W	Defines absolute target position
81	REL	4	R/W	Defines target position relative to the current position

Par ID	Name	Byte Size	Access	Description
82	POS	4	R/W	Current encoder position
83	INP	2	R/O	Reports the state of the digital inputs as 4 byte hex number.
84	IND	4	R/O	Index position captured
85	GO	0	Cmd	Start motion to the defined target position
86	FWD	0	Cmd	Start jogging forward (positive direction)
87	REV	0	Cmd	Start jogging in reverse (negative direction)
88	RESET	0	Cmd	Resets the firmware / soft restart
89	ON	0	Cmd	Enables the servo control
90	OFF	0	Cmd	Disables the servo control
91	ENABLE	0	Cmd	Enables the PWM amplifier (inverter)
92	DISABLE	0	Cmd	Disables the PWM amplifier (inverter)
93	STOP	0	Cmd	Stop motion smoothly (with the programmed deceleration)
94	ABORT	0	Cmd	Stops motion abruptly (with the maximum deceleration)
95	HOME	0	Cmd	Starts the execution of the Home procedure.
96	ALIGN	0	Cmd	Starts the execution of the Phasing procedure
97	VER	4	R/O	Reports the current firmware version
98	OUT1	2	R/W	Controls output #1
99	OUT2	2	R/W	Controls output #2
100	PWM	2	R/W	Output voltage set as PWM duty cycle (32767 = 100%). Requires that the servo control is turned off.
101	IQPKT	2	R/W	Output voltage to generate the Quadrature current component of the motor flux.
102	IDPKT	2	R/W	Output voltage to generate the Direct current component of the motor flux.
103	CH1	2	R/W	Specifies data to be recorded on channel #1 of the data recorder
104	CH2	2	R/W	Specifies data to be recorded on channel #2 of the data recorder
105	CH3	2	R/W	Specifies data to be recorded on channel #3 of the data recorder
106	CH4	2	R/W	Specifies data to be recorded on channel #4 of the data recorder
107	TRACE	2	R/W	Initialize new data recording session
108	PLAY	2	R/W	Reports recorded data
109	PLIMIT	2	R/W	I2T Protection Threshold Limit
110	PTIME	2	R/W	I2T Protection Time Span
111	GEARIN	2	R/W	Specifies the input number of a gear box transmission ratio that defines the electronic gearing ratio.
112	GEAROUT	2	R/W	Specifies the output number of a gear box transmission ratio that defines the electronic gearing ratio.
113	SETUP	2	R/W	Starts procedure to identify the proper mapping of the motor windings and the hall sensors.

Par ID	Name	Byte Size	Access	Description
114	PINVERT	2	R/W	Specifies if the encoder position feedback should be inverted: 0 – no inversion, 1 = inverted
115	SAVE		Cmd	Saves the persistent parameters to the external Flash memory
116	RESTORE		Cmd	Restores the persistent parameters from the external Flash memory
117	ETYPE	2	R/W	Encoder type: 0 = incremental, 1 = EnDat, 2 = BiSS, 3 = FA-Coder, 4 = A-format
118	EID	4	R/W	Encoder ID:
119	EADDR	2	R/W	Encoder EEPROM address
120	EDATA	2	R/W	Encoder EEPROM data to be stored at or retrieved from the address defined by the EADDR variable
121	EBAUDRATE	2	R/W	Encoder communication baud rate
122	ESTATUS	2	R/W	Absolute Encoder status
123	ADDR	2	R/W	Module address used by the packet host communication protocol
124	GROUP	2	R/W	Module group used by the packet host communication protocol
125	TBSIZE	2	R/O	Trace buffer size. Informs the host for the maximum number of samples that can be reported by the data recorder.
126	WMARK	2	R/W	PVT Buffer Watermark – defines the number of slots in the PVT which need to be occupied before watermark warning flag is raised. This is required to properly synchronize the streaming of new PVT points by the host computer.
127	QKD	2	R/W	Proportional Gain in the Quadrature current control loop
128	VKP	2	R/W	Proportional Gain in the velocity velocity loop
129	VKI	2	R/W	Integral Gain in the velocity control loop
130	VKD	2	R/W	Differential Gain in the velocity control loop
131	ELVOLT	4	R/W	Inverter bus voltage Low voltage detection threshold. value is 12bit AD value.
132	EHVOLT	4	R/W	Inverter bus voltage overvoltage detection threshold. value is 12bit AD value.
133	EWPOSMIN	4	R/W	Position abnormality Min threshold
134	EWPOSMAX	4	R/W	Position abnormality Max threshold
135	EOVS	4	R/W	Over speed threshold. value is (Position*65535)/100us.
136	EWOV	4	R/W	Instructed speed difference threshold. value is (Position*65535)/100us. make it abnormal after continuous detection time (fixed 5 sec).
137	EEMP	4	R/W	PVT Buffer EMPTY Threshold. value is the number of times.
138	EOVTEMP	4	R/W	Temperature anomaly threshold. value is 12bit AD value.
139	ERRMASK	4	R/W	Abnormal state mask. value of ErrMsk in hexadecimal.
140	EVOLT	4	R/O	Bus voltage display. value is 12bit AD value.
141	EQUERY	4	R/O	Abnormal status indication. value of ErrSts in hexadecimal.
142	ERESET	0	Cmd	Abnormal status reset.

Par ID	Name	Byte Size	Access	Description
143	EOVC	4	R/W	Overload (current) threshold. value is 12bit AD value.
144	ETEMP	4	R/O	Current temperature. value is 12bit AD value.

Appendix B Packet Communication Protocol Commands

Packet Codes:

Packet Code	Command Description	Req Packet Byte Size	Reply Packet Byte Size
0	Get Module Data	5	7
1	Data Report Request	7	Variable
2	Function Call	6	3
3	Initialize PVT Mode	6	3
4	Stream Position/Velocity set points	5 + points * 8	3
5	Set Parameter (16-bit or 32-bit)	8 or 10	3
6	Get Parameter	6	5 or 7
7	Get Trace Buffer	6	Variable
15	Set Communication Baud Rate	6	3

Reply codes:

Reply Code	Description
0	Command Accepted
1	Invalid Packet Code
2	Invalid Parameter Index
3	Invalid Data Size / Format
4	Parameter Value Out of Range
5	Invalid Request - Attempt to set a Read-Only variable
6	Interlock Active
7	Controller Fault
8	Invalid State
9	Buffer Overflow
10	Execution Error
255	Checksum error

B-1 Get Module Data (Packet Code: 0)

This packet type is only used during initialization to identify the type of the module and the version of the packet protocol. This allows the host to adapt to different types of modules and their versions as long as they conform with the format of this packet.

Header			Payload	
Start	Address	Length	Pkt Code	Check Sum
0xAA	0x01	0x00	0x00	0x01

Return Packet – Long Parameter Value

Reply Code	Long Parameter Value (4 bytes)				Check Sum
0x00	B0	B1	B2	B3	X

B0 – Protocol Version

B1 – Module type

B2 – Firmware Version (Minor Number)

B3 – Firmware Version (Major Number)

B-2 Status Report Request (Packet Code: 1)

This packet type requests report for the most commonly needed controller parameters. The report content is defined by 16-bit value. This command is dedicated for the polling of the controller status by the host periodically. The report content may change based on the context of the currently executed operation.

Header			Payload			
Start	Address	Length	Pkt Code	Request		Check Sum
0xAA	0x01	0x02	0x01	X	X	X

Request Word Definition

Request Bit #	Data Description	Data Byte Length
0 (LSB)	Current Position	4
1	Current Status	2
2	Position Error	2
3	Current Consumption	2
4	Hall Inputs	1
5	Digital Inputs	1
6	PVT FIFO Buffer Fill	1
7	Scanning Data FIFO Buffer Fill	1
8 ... 15	Reserved	N/A

Return Packet – Variable Length Data

Reply Code	Data 1	...	Data N	Check Sum
0	B0	...	Bn	X

The table below describes the purpose of the bit fields of the status word returned when Current Status report is requested.

Bit #	Bit Mask	Description
0 (LSB)	0x0001	Motion completed
1	0x0002	Servo control is on
2	0x0004	Power amplifier is enabled
3	0x0008	Position is captured
4	0x0010	Absolute Encoder Fault
5	0x0020	Absolute Encoder Warning
6	0x0040	PVT FIFO buffer depth is below the defined Watermark
7	0x0080	Phasing (Alignment) is completed
8	0x0100	Reserved
9	0x0200	Current overload
10	0x0400	Inhibit Input Triggered
11	0x0800	PVT Buffer depleted
12	0x1000	Overheating
13	0x2000	Inverter (Amplifier) Fault
14	0x4000	Position Error Exceeded
15 (MSB)	0x8000	Position Counter Wraparound

B-3 Function Call (Packet Code: 2)

No argument

Header			Payload		
Start	Address	Length	Pkt Code	Func ID	Check Sum
0xAA	0x01	0x01	0x02	X	X

The function ID is defined along with the variable IDs in the table at the end of the chapter.

Return Packet – Status

Reply Code	Check Sum
0	0

B-4 Initialize PVT Stream (Packet Code: 3)

No argument

Header			Payload		
Start	Address	Length	Pkt Code	Time Slice	Check Sum
0xAA	0x01	0x01	0x03	X	X

The function ID is defined along with the variable IDs in the table at the end of the chapter.

Return Packet – Status

Reply Code	Check Sum
0	0

B-5 Stream PVT Data (Packet Code: 4)

Header			Payload																		
Start	Address	Length	Pkt Code	Points Count	Point 1 Position				Point 1 Velocity				Point N Position				Point N Velocity				Check Sum
0xAA	0x01	8*N + 1	0x04	N	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		

Return Packet – Status

Reply Code	Check Sum
0	0

B-6 Set Parameter (Packet Code: 5)

This packet type sets a new value to the defined parameter. The parameter could be either 16 or 32-bit value. The size of the value is documented in the table of all controller parameters at the end of the chapter.

B-6.1 Set 16-bit Parameter

Header			Payload				
Start	Address	Length	Pkt Code	Par Code	Parameter Value		Check Sum
0xAA	0x01	0x03	0x05	X	B0	B1	X

B0 – LSB

B1 - MSB

B-6.2 Set 32-bit Parameter

Header			Payload						
Start	Address	Length	Pkt Code	Par Code	Parameter Value				Check Sum
0xAA	0x01	0x05	0x05	X	B0	B1	B2	B3	X

B0 – LSB

B3 - MSB

Return Packet – Status

Reply Code	Check Sum
0	0

B-7 Get Parameter (Packet Code: 6)

This packet requests the return of a controller parameter. The length of the reply depends on the specific parameter. The table with all parameters and their sizes is included at the end of the chapter.

Header			Payload		
Start	Address	Length	Pkt Code	Par Code	Check Sum
0xAA	0x01	0x01	0x06	X	X

Return Packet – Short Parameter (2 bytes)

Reply Code	Short Parameter Value (2 bytes)		Check Sum
0	B0	B1	X

B0 – LSB

B1 – MSB

Return Packet – Long Parameter Value

Reply Code	Long Parameter Value (4 bytes)				Check Sum
0	B0	B1	B2	B3	X

B0 – LSB

B3 – MSB

B-8 Report Trace Buffers Content (Packet Code: 7)

Header			Payload		
Start	Address	Length	Pkt Code	Request Byte	Check Sum
0xAA	0x01	0x01	0x07	B0	X

Buffer Definition – Byte B0 – the 4 least significant bits define the channel buffer to be reported:

B0:3-B0:0	Buffer to be reported
0	Channel #1
1	Channel #2
2	Channel #3
3	Channel #4

The most significant bit defines the data format – 16-bit or 32-bit data per sample:

B0:7	Data size
0	Data reported as 16-bit values
1	Data reported as 32-bit values

Reply packet content is defined by the Request Byte bit fields.

Return Packet – Variable Length Data

Status	Data 1	...	Data N	Check Sum
S0	B0	...	Bn	X

B-9 Set Communication Baud Rate (Packet Code: 15)

Header			Payload		
Start	Address	Length	Pkt Code	Baud Rate Code	Check Sum
0xAA	0x01	0x01	0x0F	B0	X

Return Packet – Status

Reply Code	Check Sum
0	0

Baud Rate Code Definition

Code	Baud Rate [Kbps]
0	115.2
1	230.4
2	250
3	460.8
4	500
5	921.6
6	1000
7	1250
8	1500

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Mar.3, 2023	-	First edition issued
1.01	Aug.31, 2023	P.6	2. Operating Environment Change of reference.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

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

2. Processing at power-on

The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.

3. Input of signal during power-off state

Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.

4. Handling of unused pins

Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.

5. Clock signals

After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.

6. Voltage application waveform at input pin

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).

7. Prohibition of access to reserved addresses

Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.

- Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the EU and other countries. All rights reserved.
- Ethernet is a registered trademark of Fuji Xerox Co., Ltd.
- IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers Inc
- EtherCAT® and TwinCAT® are registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.
- A-format is a trademark of the Nikon Corporation.
- BiSS is a registered trademark of iC-Haus GmbH.
- EnDat is a registered trademark of Dr. Johannes Heidenhain GmbH.
- FA-CODER is a registered trademark of Tamagawa Seiki Co., Ltd.
- HIPERFACE DSL is a registered trademark of SICK AG.
- Additionally all product names and service names in this document are a trademark or a registered trademark which belongs to the respective owners.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.

7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.