REM Switch Software Driver User Guide

INTRODUCTION

This user guide is an overview of the real-time Ethernet multiprotocol (REM) switch software driver usage. For details regarding available functions and their parameters, see the header files referenced in this user guide.

The REM switch driver is designed to provide a standard, protocol independent interface (**/Common/inc/REMS_Standard.h**) used for initialization, interrupt management, timer management, and protocol independent packet transmission and receiving. Additional functionality is accessed through protocol specific interfaces, which are designed to support the software application-level stack of a particular industrial Ethernet protocol, such as **REMS_PROFINET.h**.

The timer functionality provided by the REM switch is synchronized via an internal precision timer and can be used to capture external events or generate signals that are synchronized to a protocol specific timing function, such as precision time control protocol (PTCP), IEEE 1588, and EtherCAT distributed clock, among others. Timer functions include the following:

- Input capture, which time stamps a rising or falling edge on an external signal.
- Output compare, which generates an edge on a chip output at a programmable time.

The REM switch has many features dedicated to the protocol specific operation, including multiple transmit and receive queues, internal timer resources, and various interrupt events. The driver software manages these resources internally.

It is assumed that the user is familiar with the REM switch hardware and has reviewed the fido5100/fido5200 data sheet.

TABLE OF CONTENTS

REVISION HISTORY

3/2024—Rev. L to Rev. M

Changed Master to Leader and Slave to Follower (Throughout)....1
Changes to Introduction Section....1
Changes to Build Environment Section....5
Changes to Interrupts Section....7
Added Handle Link Change Interrupt Section....9
Changes to PROFINET Section....13
Changes to Synchronization (PTCP) Section....13
Changes to PROFINET Connection Establishment Section....15
Changes to RT Class 1 Connection Establishment Section....15
Changes to RT Class 3 Connection Establishment Section....16
Added IO Data Handling Section....16
Added PROFINET Interrupt Handling Section, Table 3, and Table 4; Renumbered Sequentially....19
Added Interrupts for One or Multiple Connections Section....22
Added Acyclic Message Reception Section....22

TABLE OF CONTENTS

CONFIGURATION

Prior to initialization or use, certain system requirements must be met for the REM switch driver to operate properly. A complete list of system requirements is detailed in Table 1.

Table 1. System Requirements

Driver Component	Requirement
Hardware	32-bit host processor or better
	50 kB to 100 kB read only memory (ROM)[1]
	8 kB random access memory (RAM)
Compiler	16-/32-/64-bit integer support
Software Libraries	
stdint.h	int8_t/uint8_t
	int16_t/uint16_t
	int32_t/uint32_t
time.h	struct timespec
string.h	memcpy()
	memset()
stdlib.h	abs()

[1] The fido1100/gcc based system uses 46 kB of ROM.

DRIVER CONTENTS

The REM switch driver is divided into the three subdirectories described in the Common section, the Porting section, and the Protocol Specific section.

Common

The common directory includes low level portions of the driver, as well as common types of functions, such as static table support and dynamic table management. Files of interest to the user are as follows:

- **inc/REMS_Standard.h**, which is an interface to standard switch functions.
- **inc/REMS_Error.h**, which contains error codes returned by driver routines.
- **inc/REMS_DynamicTable.h**, which provides management of the switch dynamic forwarding table.
- **inc/REMS_StaticTable.h**, which provides management of the switch static forwarding table.

Porting

The porting directory contains the files that must be modified to support a particular hardware platform. Nothing in this directory needs to be available to the application layer.

Protocol Specific

In the protocol specific subdirectory, a separate directory is provided for each supported protocol. Any given application depends on a single protocol. Available protocols are as follows:

- PROFINET. Only the main PROFINET header must be accessed by the software application levels of the application. **inc/REMS_PROFINET.h** provides setup and operation of PROFINET functionality.
- EtherNET/IP. Only the main EtherNet/IP header must be accessed by the software application levels of the application. **inc/REMS_EtherneIP.h** provides setup and operation of Ethernet/IP functionality.
- Modbus/transmission control protocol (TCP). Only the main Modbus/TCP header must be accessed by the software application levels of the application. **inc/REMS_ModbusTCP.h** provides setup and operation of Modbus/TCP functionality.
- EtherCAT. Two EtherCAT header files must be accessed by the software application levels of the application, as follows:
  - **inc/REMS_ECATHw.h** provides setup and operation of REM switch as an EtherCAT follower controller.
  - **inc/REMS_ECATinternals.h** sets the EtherCAT distributed clock synchronization offset parameter.

CONFIGURATION

All other header and source files are specific to the driver operation and organization and do not need to be accessed by software application levels of the application.

BUILD ENVIRONMENT

When building the REM switch driver, the following directories must be on the included path:

- **REMS_Driver/Common**
- **REMS_Driver/Porting**
- **REMS_Driver/<ProtocolSpecificDirectory>**

<ProtocolSpecificDirectory> is the name of one of the supported protocols, such as PROFINET. Build a single protocol for a given project. For example, it is not possible to build a driver that supports both PROFINET and EtherCAT. When building the rest of the application using the REM switch driver, only the common and protocol specific subdirectories must be available.

Some parameters are provided to enable debugging features and additional checks in the driver code. These parameters are listed as follows:

- REMS_ENABLE_DEBUG. If defined, this parameter enables the error level debug macros embedded in the code. If REMS_ENABLE_DEBUG is set to 1, this parameter also enables informational printouts. For example, if the symbol is set on the compiler command line, the command code may look as follows:
  - -D REMS_ENABLE_DEBUG (generates error printouts)
  - -D REMS_ENABLE_DEBUG = 1 (generates error and informational printouts)
- REMS_PARAMETER_CHECKS. If defined, this parameter causes the code to perform detailed checks on function parameters. Defining this parameter on a command line may look like -D REMS_PARAMETER_CHECKS.

PORTING

Because the driver has no dependencies on any operating system resources (such as no threading and semaphores), porting is limited to defining how the host processor communicates with the REM switch and some debugging options. All porting related code is located in the porting directory and found in the **/Porting/inc/REMS_Port.h** and **/Porting/src/REMS_Port.c** files. The porting/examples directory contains versions of these files that are specific to various processors.

REMS_Port.h

The REMS_DEBUG(), REMS_INFO(), and REMS_EVENT() debug macros may require platform specific customization. When enabled, these macros are used by driver code to display error and warning events. Calling printf() implements these macros. If printf() is not available, other logging mechanisms can be used.

The following parameters in **REMS_Port.h** define the hardware environment in which the driver operates:

- REMS_32_BIT_BUS. Define this parameter if the interface from the host processor to the REM switch is 32 bits wide. Otherwise, the driver assumes the interface is 16 bits wide.
- REMS_LITTLE_ENDIAN_HOST. Define this parameter if the host processor uses little endian byte ordering for memory accesses. Otherwise, the driver assumes the host processor uses big endian byte ordering.
- REMS_BASE_ADDRESS. This parameter must be set to the base address at which the REM switch is accessed, or the location of the external chip select in the memory map.
- REMS_ADDR_SHIFT. This parameter indicates how far the address values are shifted depending on the size of the host processor data bus. The default operation of the driver is to work with byte addressable memory (REMS_ADDR_SHIFT = 0). If the memory bus addresses 16-bit words, set REMS_ADDR_SHIFT to 1. If the memory bus addresses 32-bit words, set REMS_ADDR_SHIFT to 2. If using separate address and data buses to communicate to REM, coordinate the setting of this parameter with the alignment of the address bus to address the input of the REM. If using a multiplexed address or data bus, this parameter can be used to adjust for the operation of the user bus processor.
- _SWAPL_() and _SWAPS_(). Use these macros to swap bytes in limited cases when reading or writing packets. These macros are only used for control words, packet sizes, and time stamps, but not all packet data. Because the REM switch uses big endian internal organization, these macros have no effect on a big-endian host. If user processor has hardware support for endian swapping, rewriting the macros is possible.

CONFIGURATION

REMS_Port.c

Typically, only the REMS_ReadBlock() and REMS_WriteBlock() functions in this file must be modified. REMS_ReadBlock() is used by the driver to read packet data from a first in, first out (FIFO) queue and REMS_WriteBlock() is used to write packet data to a FIFO queue. The sample processor example code is a simple C-language implementation for a 16-bit data bus. The sample processor can be easily adapted to a 32-bit bus by changing the short pointers to long pointers. If direct memory access (DMA) resources are available on the user processor, they can be used within REMS_ReadBlock() and REMS_WriteBlock().

USING THE DRIVER

STANDARD INTERFACE

Basic, protocol independent functionality of the REM switch is accessible through **REMS_Standard.h**.

Driver Initialization

When initializing the REM switch, first call REMS_StdInit(). REMS_StdInit() checks basic communication between the host processor and the REM switch, loads the protocol specific configuration to the device, defines the communication interface for the physical Ethernet layers (PHYs) (media independent interface (MII) and reduced media independent interface (RMII)), and identifies which of the REM switch interrupt outputs are associated with events of different priorities. If REMS_StdInit() returns REMS_OK, proceed to the next step. If REMS_OK is not returned, there is a communication error to the REM switch.

After the REM switch is initialized, the media access control (MAC) addresses for the REM switch must be set. Either one or three MAC addresses are required, depending on the protocol. PROFINET requires a host MAC address for each port, whereas other protocols require a single host address. Call REMS_StdSetMacAddress() to call the MAC addresses.

Interrupts

The REM switch supports up to three separate interrupt lines. While fewer lines may be used, use of all three interrupt lines yields the best performance on protocols with critical timing. The interrupt events supported by the device and driver are defined in **REMS_Standard.h** in the REMS_StdIntEvent_t enumerated type. Additional interrupt events from the device are handled directly by the driver. The REM switch hardware and driver are compatible with either edge triggered or level sensitive interrupts.

Interrupts are configured using the following routines:

- REMS_StdAssignInterrupt() determines the line and priority for an interrupt event.
- REMS_StdEnableInterrupt() enables an interrupt event.
- REMS_StdDisableInterrupt() disables an interrupt event.

Hardware events in REMS_StdIntEvent_t are available for use. Protocol specific events are typically generated by the driver based upon the occurrence of events more complex than those in REMS_StdIntEvent. Protocol specific events are already enabled or disabled and are already assigned to a particular interrupt line by the driver when firmware is loaded.

To define interrupt handlers for the various priorities, use a structure similar to the following. The specific handler syntax varies depending on user environment.

```
/*** Handle REM Switch Interrupt Line 0.  ***/

__attribute((interrupt, section (".rem"))) void LEASHstandardExternalInt0()
{
  volatile unsigned long ack;
  /* acknowledge the interrupt */
  ack = *FIDO_INTCONTROLCH0;

  HandleIntREMS(REMS_Int_Line_0);
}
```

Interrupt Line 1 and Interrupt Line 2 use similar handler syntaxes, with the difference being that Line 1 and Line 2 take the place of Line 0. The main handler routine functions regardless of how interrupt events are allocated across the interrupt lines and may appear as follows:

```
static void HandleIntREMS(REMS_IntLine_t line)
{
  REMS_stdIntEvent_t event;
  unsigned short status;

  /* Have driver read interrupt events from REM and queue them up locally */
```

USING THE DRIVER

```
REMS_StdEvaluateInterrupt(line);
do {
  /* read events from the local queue one at a time */
event = REMS_StdGetNextEvent(line);

switch (event) {
case REMS_StdInt_Port_1_LinkChange:
  port = REMS_enetPort_1;
  AddToQueue(linkManagementQueue_g, &port, WAIT_FOREVER);
  break;

case REMS_StdInt_Port_2_LinkChange:
  port = REMS_enetPort_2;
  AddToQueue(linkManagementQueue_g, &port, WAIT_FOREVER);
  break;

case REMS_StdInt_PktReady:
  /* handle a packet received on the low priority queue,
   * pass it to the TCP/IP stack */
  REM_ReceivePacket();
  break;

case REMS_StdInt_Capture_0:
  /* handle timer capture events … */
  break;
case REMS_StdInt_Capture_1:
  break;
case REMS_StdInt_Capture_2:
  break;
case REMS_StdInt_Capture_3:
  break;
  /* handle timer output compare events */
case REMS_StdInt_Compare_0:
  break;
case REMS_StdInt_Compare_1:
  break;
case REMS_StdInt_Compare_2:
  break;
case REMS_StdInt_Compare_3:
  break;
  /* Handle periodic timer events (TCU) */
case REMS_StdInt_TimerControl_0:
  break;
case REMS_StdInt_TimerControl_1:
  break;
case REMS_StdInt_TimerControl_2:
  break;
case REMS_StdInt_TimerControl_3:
  break;

/* Protocol-specific interrupt events */    /*...*/
default:
break;
}
```

```
} while (event != REMS_Int_None); /* handle all outstanding events */
```

Handle Link Change Interrupt

This syntax provides an example of how to manage REMS_StdInt_Port_1_LinkChange and REMS_StdInt_Port_2_LinkChange interrupts. In the previous example, the link change event and port number were added to a queue. In this separate thread, the queue is evaluated, and the necessary actions for each event and port are taken:

```
queue linkManagementQueue_g;

/* Used to save link event data and provide to IP stack;
 * interface ID, link up/down, port of link event, interface speed, duplex mode
 */
linkEventInfo_t linkEvent_g

/* Enum; provides speed and duplex mode per port */
linkState_t portSpeedDuplex_g[NUM_DEVICE_PORTS];

/* Used to save PHY config parameters for external use; auto-negotiation status, speed, du▶
plex mode, mdi crossover settings */
phyConfig_t phyConfig_g[]

void HandleLinkChange(void *param)
{
  EthLinkStatus_t linkUp;
  REMS_SpeedDuplex_t REMSspeedDuplex;
  REMS_CommonEnetPort_t port;
  uint8_t partnerAutoNegState;

  do{
    RemoveFromQueue(linkManagementQueue_g, &port, WAIT_FOREVER);

    // set event notification interface index
    linkEvent_g.interfaceIdx = ifHandle_g;
    // determine the PHY address and type
    if(port == REMS_enetPort_1){
      linkEvent_g.port = macPort0;
    }else if(port == REMS_enetPort_2){
      linkEvent_g.port = macPort1;
    }

    // Get the link status from REMS
    linkUp = REMS_StdGetLinkState(port);

    // Perform an edge detect on link up/down
    if((linkUp == LINK_UP) &&
       (portStatus_g[port-1] == LINK_DOWN))
    {
      /* Link went up */
      linkEvent_g.link = true;

      // Get the speed and duplex from the PHY

      // Get the link partner auto-negotiation state
```

USING THE DRIVER

```
    if((partnerAutoNegState == LINK_PARTNER_FORCED) &&
       (phyConfig_g[port-1].autoNegEnable)) {
      /*Set Phy Duplex mode; For PROFINET, fall back to full duplex mode if auto-negotiation fails*/

      // Save full duplex mode at port status variable
      // portSpeedDuplex_g[port-1] = duplexMode
    }
    // Tell REMS what the new link settings are
    switch(portSpeedDuplex_g[port-1]){

    /*...*/
    case LinkState100BaseTxFd:
      REMSspeedDuplex = REMS_100_Full_Duplex;
      linkEvent_g.duplex = phyDuplexModeFull;
      linkEvent_g.speed = phySpeed100;
    break;

    /*...*/
    }

    // Set REMS' speed and duplex
    REMS_StdSetSpeedAndDuplex(port, REMSspeedDuplex);
    // Set the port state to forwarding
    REMS_StdSetPortState(port, REMS_PORT_FORWARDING);

    // Update PHY configuration settings if auto-negotiation enabled
    if (phyConfig_g[port-1].autoNegEnable)
    {
      phyConfig_g[port-1].speed = linkEvent_g.speed;
      phyConfig_g[port-1].duplexMode = linkEvent_g.duplex;
    }

    // Store the port link status for the next go 'round and so others can see it
    portStatus_g[port-1] = linkUp;

    // Last thing publish the link up event to all other subsystems
    PublishEvent(LINK_UP);

  }else if((linkUp == LINK_DOWN) &&
     (portStatus_g[port-1] == LINK_UP))
  {
    /* link went down */

    /* check opposite port to determine there is a link on either port */
    if (portStatus_g[(((port-1) == 0)? 1:0)] == LINK_DOWN)
      linkEvent_g.link = false;
    else
      linkEvent_g.link = true;

    // Read the PHY status register to unlatch the link status bit

    // Shut the port down since there is no link
    REMS_StdSetPortState(port, REMS_PORT_OFF);

    // Flush the dynamic table on link down
```

```
      REMS_FlushDynamicTable();

      // Publish the link down event
      linkEvent_g.duplex = phyDuplexUnknown;
      linkEvent_g.speed = phySpeedUnknown;

      // Update PHY configuration settings if auto-negotiation enabled
      if (phyConfig_g[port-1].autoNegEnable) {
        phyConfig_g[port-1].speed = linkEvent_g.speed;
        phyConfig_g[port-1].duplexMode = linkEvent_g.duplex;
      }

      // Store the port link status for the next go 'round and so others can see it
      portStatus_g[port-1] = linkUp;

      // Last thing publish the link down event to all other subsystems
      PublishEvent(LINK_DOWN);
    }

  }while (1);
}
```

Note that a separate thread is provided to handle the link change event, to keep runtime during interrupt service routines (ISRs) short. Also in the case for PROFINET, a call to eddf_interrupt(), which follows the publishing of link down and up events, must not be called from an ISR context.

Packet Transmission and Receiving

REMS_Standard.h provides an interface by which to transmit and receive packets over the lowest priority queue. This interface leaves the higher priority queues open for protocol specific traffic. The transmit interface allows the user to determine the port from which the packet transmits and the receive interface indicates on which port a packet was received. These interfaces, in conjunction with the static forwarding table, allow the use of protocols such as link layer discovery protocol (LLDP), rapid spanning tree protocol (RSTP), and media redundancy protocol (MRP).

As noted in the Porting section, the routines that read and write packet data to and from the REM switch can be customized to take advantage of DMA resources on the host processor.

Synchronized Timer Signals

The timer control unit (TCU) controls the external timer signals on the REM switch. These signals come directly from the REM switch. They are also synchronized directly to the network. The TCU can be called using the functions REMS_StdGetTcuTimersAvailable() and REMS_StdSetTcuIoParams(). Both functions allow the TCU to generate a timer signal that fits the needs of the system. Both functions are defined in **REMS_Standard.h**.

REMS_StdGetTcuTimersAvailable() is used to get the number of TCU timer signals that are available to the application.

REMS_StdSetTcuIoParams() allows the user to specify parameters for the timer in use. Call this function to set pulse parameters on the indicated timer channel.

ADDRESSING TABLES

Static Forwarding Table

The static forwarding table is used to manage the handling of packets with a multicast destination MAC address. The interface for managing the table is located in **REMS_StaticTable.h**. By default, the REM switch forwards any multicast packet from the receiving Ethernet port to the other

USING THE DRIVER

Ethernet port but does not forward the packet to the host processor. The user can add a specific multicast MAC address to the static table to alter the forwarding of the multicast packet.

REMS_AddStaticTableEntry() defines the rule to apply to packets with a given address, and also indicates whether the packets override port state settings.

The static forwarding table also provides functions to remove an entry and to flush all entries from the table. The number of entries in the static forwarding table varies by protocol, as shown in Table 2. The static forwarding table only applies to packets received from an Ethernet port.

Table 2. Static Forwarding Table Entries

Protocol	Number of Entries in Static Forwarding Table
PROFINET	16
EtherNet/IP	6
Modbus/TCP	8
EtherCAT	0

Dynamic Forwarding Table

The dynamic forwarding table is used to manage the handling of packets with a unicast destination MAC address. This table operates automatically and does not require input from the application. A common operation for the dynamic forwarding table is to change the aging time for entries in the table, which affects how long a forwarding rule for an address stays in the table without being refreshed by a packet received from that address. The dynamic forwarding table is also often used to flush the table on a network topology change.

These functions, along with the ability to add or delete an address in the table, are available in **REMS_DynamicTable.h**. The current dynamic table implementation allows aging time between 12 sec to 12 minutes. The dynamic forwarding table is set to 512 items by default and is not used by EtherCAT.

PROTOCOLS

At the time of this release of the driver, the supported protocols are EtherCAT, PROFINET, Modbus/TCP, Ethernet/IP, and POWERLINK. These are described in the PROFINET section, the EtherNet/IP section, the Modbus/TCP section, the EtherCAT section, and the POWERLINK section.

PROFINET

The current version of the driver supports a relative forwarder device of Version 2.43 (as of 1/11/2024 beyond this date the intent is for the device to support updated stack versions) of the PROFINET Specification, Conformance Class C.

Manage the standard setup of the TCP/IP stack as usual. LLDP and discovery and configuration protocol (DCP) packets are relayed to the host via the low priority queue, which is interfaced through **REMS_Standard.h**. The integrator must differentiate traffic as necessary. The interface for input/output (I/O) data is described in the RT Class 1 Connection Establishment section and the RT Class 3 Connection Establishment section.

The PROFINET REM switch driver also provides the ability to support system redundancy (S2). This has an implication that there must be multiple connections to the PROFINET device. The PROFINET driver can support up to four application relations (ARs)/connections.

SYNCHRONIZATION (PTCP)

PTCP line delay and synchronization traffic are handled entirely by the driver.

Most parameters are dependent on the specifics of the connection process. Ensure that the PROFINET transmit and receive delay values of the device are set at initialization time. Also, ensure that these are the same delay values advertised in the GSDML file of the device and in LLDP packets transmitted by the device. These delay values are constants for a particular hardware design, which are dependent on the PHYs chosen. The receive value represents the time between the arrival of a packet on the cable side of the receive PHY and the time stamp in the REM switch. The transmit value represents the time between the time stamp in the REM switch device and the packet being transmitted on the cable side of the PHY.

PHYs targeting the PROFINET IRT market include estimates of this timing in the respective device datasheet.

Some adjustments must be made to the PHY values to account for timing within the REM switch. The PHY transmit delay takes an additional 64ns in the REM switch, and the receive delay in the REM switches 8ns less than the PHY receive delay. Use the REMS_PnetSetDelayValues() function to provide these values to the driver.

PTCP messages are received and evaluated through EDDF_NRT_ISR_Acyc_Rcv_IFA() (see the Acyclic Message Reception section). You can also check for PTCP messages in incoming packets received through signal REMS_Int_Queue_1_Packet_Ready yourself. Check for a PTCP message with the function REMS_SyncIsSyncFrame(), and if a PTCP frame is received, evaluate it with REMS_SyncHandleMessage(). The following events can be returned:

- REMS_PnetInt_Sync indicates that the local machine is synchronized with the leader. Notify PROFINET stack and REMS driver with EDDF_SYNCSyncSet() and REMS_SynchronizationActive(), respectively.
- REMS_PnetInt_Jitter_Out_Of_Boundary indicates a loss of synchronization. Notify PROFINET stack and REMS driver with EDDF_SYNCSyncSet() and REMS_SynchronizationInactive(), respectively.
- REMS_Int_None means nothing to do.

Provide a separate thread that runs after a REMS_Int_Periodic_Timer_0 event is received through REMS driver in order to call REMS_SyncPeriodicProcessing() periodically to start line and cable delay evaluation. Evaluate the return value and check if a synchronization loss event occurred as follows:

- REMS_PnetInt_No_Sync_Message_Received notifies REMS driver as well as electronic device description file (EDDF) of the synchronization loss by calling REMS_SynchronizationInactive() and EDDF_SYNCSyncSet(), respectively.
- REMS_Int_None means nothing to do.

After the delay values are set, call REMS_SyncStartBridge(). Calling this function initializes the basic message processing and data structures for line delay calculation and synchronization. REMS_SyncStartBridge() requires two function pointers to functions acquiring and releasing a mutex/semaphore to provide exclusive access to REMS driver transmit interface.

Line delay processing takes place without intervention from the software application levels. Either based on parameters from nonvolatile memory or data provided in the connection request, call REMS_SyncStartFollower() when appropriate. This function provides the parameters necessary for the driver to validate synchronization packets and synchronize to the proper leader. At this point, the driver begins the process of synchronization.

REMS driver provides new line and cable delay values as well as diagnosis events for the PROFINET stack. Both delays and diagnosis events need to be polled after the evaluation of a PTCP message and after calling REMS_SyncStopBridge(). New delay values can be polled through REMS_SyncSignalDelayResponse() and new diagnosis events are polled through REMS_SyncDiagnosisEventAvailable(). After new delay values are available, follow these steps:

PROFINET

1. Get cable delay with REMS_SyncCableDelay().
2. Get line delay with REMS_SyncLineDelay().
3. Provide line and cable delay to PROFINET stack with EDDF_SYNCDelaySet().

If new diagnosis events are available, retrieve them with REMS_SyncGetDiagEvent(). Before forwarding the event to PROFINET stack, the returned follower state requires translation as follows:

```
#define REMS_SYNC_FOLLOWER_STATE_OFF       (0)
#define REMS_SYNC_FOLLOWER_STATE_FIRST     (1)
#define REMS_SYNC_FOLLOWER_STATE_WAIT      (2)
#define REMS_SYNC_FOLLOWER_STATE_WAIT_SET  (3)
#define REMS_SYNC_FOLLOWER_STATE_SET       (4)
#define REMS_SYNC_FOLLOWER_STATE_SYNC_IN   (5)
#define REMS_SYNC_FOLLOWER_STATE_SYNC      (6)
#define REMS_SYNC_FOLLOWER_STATE_SYNC_OUT  (7)
#define REMS_SYNC_FOLLOWER_STATE_OUT       (8)
#define REMS_SYNC_FOLLOWER_STATE_STOP      (9)
#define REMS_SYNC_FOLLOWER_STATE_STOPPING  (10)

typedef struct DiagnosisData_t {
  uint8_t source;
  uint8_t followerState;
  uint16_t receiveSyncPriority;
  uint32_t sequenceId;
  uint32_t rateInterval;
  uint8_t* leaderMacAddr;
} DiagnosisData;

DiagnosisData data;
    switch(followerState) {
      case MachineUninitialized:
      case MachineInitialized:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_OFF;
        break;
      case Machine_Idle:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_FIRST;
        break;
      case Machine_Wait:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_WAIT;
        break;
      case Machine_Wait_Set:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_WAIT_SET;
        break;
      case Machine_Async:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_SYNC_IN;
        break;
      case Machine_Sync:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_SYNC;
        break;
      case Machine_Tsync:
        data.followerState = REMS_SYNC_FOLLOWER_STATE_SYNC_OUT;
        break;
    }
```

PROFINET

To notify PROFINET stack of the diagnosis event, acquire a request block with EDDF_GlbRQBQueueRemoveFromBegin(&pDDB->SYNC.pHDB->pSYNC->DiagReqQueue, fill the parameters accordingly, and call EDDF_GlbRequestFinish(pDDB->SYNC.pHDB, pRQB, EDD_STS_OK) on the request block.

The following functions are useful to the software application levels:

- REMS_SyncGetLeaderAddr() returns the MAC address of the synchronization leader for presentation as leader in LLDP packets.
- REMS_SyncGetPeerDelays() returns the transmit and receive delays of the peer per port for presentation as leader in LLDP packets.

PROFINET CONNECTION ESTABLISHMENT

REMS_PnetConnectionStart() must be called when a connect.req frame is received, indicated by a call to the callback function PNIO_cbf_ar_connect_ind(). This function provides the driver with the MAC address, application relation (AR) type, and start up mode of the I/O controller necessary for a connection. The application then must initialize a provider protocol machine (PPM) and consumer protocol machine (CPM). The REMS_PnetPpmInsert() and REMS_PnetCpmInsert() functions are provided for PPM and CPM initialization. There is no order requirement when initializing the PPM and CPM. To start the PPM, call REMS_PnetPpmStart().

If the connection is closed for any reason, such as a consumer watchdog timeout, the PPM and CPM must be removed using the REMS_PnetPpmRemove() and REMS_PnetCpmRemove() functions, respectively. CPM and PPM initialization, startup, and removal are currently handled by the PN stack via the EDDF in their respective files (**eddf_crt_cons.c** and **eddf_crt_prov.c**). In order to provide the PN stack with new IP parameters, use the PNIO_change_ip_suite() function, when your IP stack provides new parameters.

The PROFINET stack also needs to be notified of link change events (see the Handle Link Change Interrupt section). To notify the PN stack, provide something similar to the following:

```
void HandleEvent(int eventIndex){
  switch (event) {
    /* ... */
    case (LINK_UP):
    case (LINK_DOWN):
      pDDB = &g_EDDF_coreDev;
      if (!pDDB->Core.Isr.Active) {
        // Save link indication, if PN Stack is not yet ready
        PNET_linkIntFired_g = 1;
      }
      else {
        eddf_interrupt(pDDB->hDDB, EDDF_INT_STATUSCHANGE);
      }
    break;

    /* ... */
  }
}
```

Note that calls to eddf_interrupt(), which follow the publishing of link down and up events, or reception of NRT packets from Queue 1 (see the Acyclic Message Reception section) may not be made from an ISR context. Therefore, ensure to provide one or more separate threads, from which the call to eddf_interrupt() is issued.

RT CLASS 1 CONNECTION ESTABLISHMENT

For a PROFINET RT Class 1 connection, there are no additional parameters required. The application must only call REMS_PnetUserDataValid() when the device is ready to send the ApplicationReady.req frame, for example, after the PN CPU is finished with parameterization of the PN device, indicated by PNIO_cbf_param_end_ind(), if parameter MoreFollows is set to false. Additionally, a call is required after the PN CPU issues a ready for input update event (PNIO_cbf_ready_for_input_update_ind()).

When support for legacy startup mode is required, check the startup mode via startMode_g in PNIO_cbf_ar_connect_ind() and issue a call to REMS_PnetUserDataValid().

PROFINET

RT CLASS 3 CONNECTION ESTABLISHMENT

For a PROFINET RT Class 3 connection, the application must call REMS_IrBeginEnddata_PortAssignment() for both ports when the port data input register (PDIR) data index Write.req frame is received. Then, the application can call REMS_pnetUserDataValid() when the device is ready to send the ApplicationReady.req frame. The application also manages the RT_Class_3 port state machine. Use the REMS_pnetSetPortRedState() function to set the switch port state to off, up, or run to match the LLDP frames. Finally, call REMS_pnetReadyForRTClass3() when the device is fully synchronized and ready for a Class 3 connection.

IO DATA HANDLING

The IO management (IOM) can be used to set up buffers for the communication relation and communicate them to the PN stack and the REMS driver. PNIOD_PLATFORM_SLATER or PNIOD_PLATFORM_ECOS_FIDO defines are used to set up the calls to the functions defined for use with fido5100/fido5200.

An example of the IOM implementation can be found in **iom_swif.c**, which can be adjusted for the necessary platform.

IOM can handle setup, initialization, and access to the buffers required. iom_swif_init() and iom_swif_allocate_iocr_memory() are used to initialize the buffers and provide the necessary sizes for each relevant AR.

The REMS driver currently supports four different ARs. Therefore, four provider buffers can be set up. Consumer buffers are supposed to be doubled buffers, to allow for swapping the active, to be written to. While the REMS driver writes received data to one buffer, the PN stack is able to work with the second provided buffer.

In general, the REMS driver is provided with buffers for cyclic data exchange during REMS_PnetCpmInsert() and REMS_PnetPpmInsert() calls. Provide both CPM and PPM with the buffers initialized by IOM. The consumer buffer is expected to be swapped after reception of each cyclic data frame (see the REMS_PnetInt_ReceivedRtDataX section).

iom_swif_consumer_lock() and iom_swif_provider_lock() are used throughout the PN stack to access the locations of CPM and PPM buffers. The locking feature of the critical sections of these functions can be ignored as of now.

It is important to make sure to provide buffers with a minimum size of 40 bytes. Smaller buffer sizes lead to mangled messages as the REMS driver expects the minimum size of 40 bytes for REMS_WritePacket().

To safely read out CPM data, retrieve, for example, the primary AR ID and retrieve the relevant buffer with a call to iom_consumer_lock(). The PROFINET IO device user interface for the customer application (PNPB) provides the offset of the cyclic data inside the IOM buffer, as well as data length. Also during read out, provide the PNPB with the current application protocol data unit (APDU) status as follows:

```
int32_t PNET_ReadItem(int32_t itemID, void *data_p)
{
  int32_t rv = (PNIO_NOT_OK); // IO data invalid
  LSA_UINT8* cpmDataBuf_p;
  iom_apdu_status_t* apduStatus_p;
  PNIO_EXP_SUB*  pExpSub;
  LSA_UINT16 arIdx;

  //get primary AR
  arIdx = iom_swif_get_session_prim_ar_idx();
  if (itemID >= PnpbExp[arIdx].NumOfPluggedSub)
    // Controller hasn't plugged in the submodule associated with this item
    // so report back there's no valid output data for it
    return (PNIO_NOT_OK);

  // get pointer to IOCR buffer and APDU status (including cycle counter and status bits)
  iom_consumer_lock((void*)&cpmDataBuf_p, &apduStatus_p, arIdx);

  if (cpmDataBuf_p == NULL)
  {
    OsExitX (OS_MUTEX_PNPB_IO);
    OS_INSTRUMENT_USER_STOP(0xAA, 5678);
```

```
    return (PNIO_NOT_OK);

  }

  // Provide PNPB with APDUStatus
  OsMemCpy(&PnpbExp[arIdx].LastApduStat, apduStatus_p, 4);

  pExpSub = &(PnpbExp[arIdx].Sub[itemID]);
  if (pExpSub->OwnSessionKey)
  { // AR is submodule owner
    if (pExpSub->IoProp & PNIO_SUB_PROP_OUT)
    {
      if (pExpSub->isPlugged)
      {
        // Readout data from IOM buffer to provided data pointer
        OsMemCpy(data_p, cpmDataBuf_p + pExpSub->Out.data_offset, pExpSub->Out.data_length);
        rv = (PNIO_OK); // IO data valid
      }
      else
      {
        pExpSub->Out.iocs_val = PNIO_S_BAD;
      }
      /* save remote producer state */
    }
    if (pExpSub->IoProp & PNIO_SUB_PROP_IN)
    {
      /* save remote consumer state */
      pExpSub->In.iocs_val = *(cpmDataBuf_p + pExpSub->In.iocs_offset);
    }
  }
  else
  { // AR is not submodule owner
    if (pExpSub->IoProp & PNIO_SUB_PROP_IN)
    { // save remote consumer state
      pExpSub->In.iocs_val = PNIO_S_BAD;
    }
  }

  // **** free pointer to IOCR buffer ***
  iom_consumer_unlock(arIdx);
  return rv;
}
```

Similarly, proceed when writing data items, and additionally provide items with their current IO provider status (IOPS) value as follows:

```
int32_t PNET_WriteItem(int32_t handle, void *data_p)
{
  LSA_UINT8* ppmDataBuf_p;
  PNIO_EXP_SUB* pExpSub;
  LSA_UINT16 arIdx;

  arIdx = iom_swif_get_session_prim_ar_idx();

  if (handle >= PnpbExp[arIdx].NumOfPluggedSub)
    // Controller hasn't plugged in the submodule associated with this item
```

```
    // so just return OK since there's no buffer slot for it
    return (PNIO_NOT_OK);

  iom_provider_lock((void*)&ppmDataBuf_p, arIdx);
  pExpSub = &(PnpbExp[arIdx].Sub[handle]);

  //provide io data and iops from input submodule
  if (pExpSub->OwnSessionKey && pExpSub->isPlugged)
  {
    if (pExpSub->IoProp == PNIO_SUB_PROP_NO_DATA)
    { // ** submod has no IO data **
      // *** update (input-)provider state in input frame ***
      if ((!pExpSub->IsWrongSubmod) && pExpSub->ParamEndValid)
        *(ppmDataBuf_p + pExpSub->In.iops_offset) = pExpSub->In.iops_val;
      else
        *(ppmDataBuf_p + pExpSub->In.iops_offset) = PNIO_S_BAD;
    }
    else
    {
      if (pExpSub->IoProp & PNIO_SUB_PROP_IN)
      {  // ** submod has input data **
          // *** update (input-)provider state in input frame ***
        if ((!pExpSub->IsWrongSubmod) && pExpSub->ParamEndValid)
        {
          OsMemCpy(ppmDataBuf_p + pExpSub->In.data_offset, data_p,
                   pExpSub->In.data_length);
          *(ppmDataBuf_p + pExpSub->In.iops_offset) = pExpSub->In.iops_val;
        }
        else
        {
          *(ppmDataBuf_p + pExpSub->In.iops_offset) = PNIO_S_BAD;
        }
      }
      if (pExpSub->IoProp & PNIO_SUB_PROP_OUT)
      { // ** submod has out put data **
        if ((!pExpSub->IsWrongSubmod) && pExpSub->ParamEndValid)
          *(ppmDataBuf_p + pExpSub->Out.iocs_offset) = pExpSub->Out.iocs_val;
        else
          *(ppmDataBuf_p + pExpSub->Out.iocs_offset) = PNIO_S_BAD;
      }
    }
  }
  else // submod not owned or not plugged
  {
    if (pExpSub->IoProp == PNIO_SUB_PROP_NO_DATA)
    { // ** submod has no data **
      *(ppmDataBuf_p + pExpSub->In.iops_offset) = PNIO_S_BAD;
    }
    else
    {
      if (pExpSub->IoProp & PNIO_SUB_PROP_IN)
      { // ** submod has input data **
        OsMemSet((void*)(ppmDataBuf_p + pExpSub->In.data_offset), 0x00,
                 pExpSub->In.data_length);
        *(ppmDataBuf_p + pExpSub->In.iops_offset) = PNIO_S_BAD;
      }
```

```
      if (pExpSub->IoProp & PNIO_SUB_PROP_OUT)
      { // ** submod has out put data **
        *(ppmDataBuf_p + pExpSub->Out.iocs_offset) = PNIO_S_BAD;
      }
    }
  }
  iom_provider_unlock(arIdx);

  return (PNIO_OK);
}
```

NETLOAD FILTERING

The PROFINET REM switch driver provides frame filtering capabilities for net load management. All filtering is disabled by default. Two types of filtering can be applied by the REM switch PROFINET firmware. In the first type, careful filtering is applied to eliminate certain classes of frames that are not directed to the local device, such as broadcast address resolution protocol (ARP) requests that do not match the local IP address. In the second type, frames directed to the local device are dropped if the host processor is asked to process high volume of packets. In such a case, priority is given to frames necessary to maintain the PROFINET connection.

The following functions are provided for enabling and configuring these filters:

- REMS_PnetResetQueue0filterCount() resets the Q0 (lowest priority) frame counter. The first time this function is called, the driver sets a counter that is decremented with every received frame on Q0. The frame count starts at 10 frames and if the counter reaches 0, the driver starts dropping frames that do not have a source MAC address matching the controller of an established connection. The REMS_pnetResetQueue0filterCount() function must then be called periodically to reset the Q0 filter count. Call this function in the lowest priority thread to ensure all threads are serviced under heavy net load. Under normal conditions, the counter never reaches 0 and no frames are dropped.
- REMS_PnetSetLldpFilter() sets the MAC address of the neighbor port for a given device port. This function adjusts the low priority filtering to allow all frames from another MAC address. This function allows neighbor LLDP frames through, even if the frame counter reaches 0.
- REMS_PnetSetDcpName() sets the device name for DCP device identify request frame filtering. This function enables the filtering of all name of station identify request frames that do not match the given name of station.
- REMS_PnetSetDcpAlias() sets the alias name for DCP device identify request frame filtering. This function enables the filtering of all alias name identify request frames that do not match the given alias name.
- REMS_PnetSetArpFilter() sets the IP address for ARP filtering. This function enables the filtering of all ARP requests that do not match the device IP address.

PROFINET INTERRUPT HANDLING

For PROFINET interrupts, the events that are required are left to the application to implement. They must first be assigned using REMS_PL_AssignRemsInts() and then enabled using REMS_PL_EnableRemsInts() as shown in the following code snippets:

```
int32_t REMS_PL_AssignRemsInts()
{
  // Assign standard application layer interrupts to the desired lines
  REMS_StdAssignInterrupt(REMS_Int_Queue_3_Packet_Ready, REMS_Int_Line_2);
  REMS_StdAssignInterrupt(REMS_Int_Queue_2_Packet_Ready, REMS_Int_Line_2);
  REMS_StdAssignInterrupt(REMS_Int_Timer_Control_Int_0,  REMS_Int_Line_2);
  REMS_StdAssignInterrupt(REMS_Int_Timer_Control_Int_1,  REMS_Int_Line_2);
  REMS_StdAssignInterrupt(REMS_Int_Timer_Control_Int_2,  REMS_Int_Line_2);
  REMS_StdAssignInterrupt(REMS_Int_Periodic_Timer_0,     REMS_Int_Line_1);
  REMS_StdAssignInterrupt(REMS_Int_Queue_1_Packet_Ready, REMS_Int_Line_1);
  REMS_StdAssignInterrupt(REMS_StdInt_Port_1_LinkChange, REMS_Int_Line_0);
  REMS_StdAssignInterrupt(REMS_StdInt_Port_2_LinkChange, REMS_Int_Line_0);

  // Standard TCP/IP
```

PROFINET

```
  REMS_StdAssignInterrupt(REMS_StdInt_PktReady,          REMS_Int_Line_0);

  REMS_StdAssignInterrupt(REMS_Int_Periodic_Timer_1,     REMS_Int_Line_0);
  REMS_StdAssignInterrupt(REMS_Int_Host_Port_0,          REMS_Int_Line_0);
  return (REMS_PL_OK);
}

int32_t REMS_PL_EnableInterrupts()
{
  // Enable the desired standard REMS interrupts
  REMS_StdEnableInterrupt(REMS_StdInt_Port_1_LinkChange);
  REMS_StdEnableInterrupt(REMS_StdInt_Port_2_LinkChange);
  REMS_StdEnableInterrupt(REMS_StdInt_PktReady);
  REMS_StdEnableInterrupt(REMS_Int_Queue_3_Packet_Ready);
  REMS_StdEnableInterrupt(REMS_Int_Queue_2_Packet_Ready);
  REMS_StdEnableInterrupt(REMS_Int_Queue_1_Packet_Ready);
  REMS_StdEnableInterrupt(REMS_Int_Timer_Control_Int_0);
  REMS_StdEnableInterrupt(REMS_Int_Timer_Control_Int_1);
  REMS_StdEnableInterrupt(REMS_Int_Timer_Control_Int_2);
  REMS_StdEnableInterrupt(REMS_Int_Periodic_Timer_0);
  REMS_StdEnableInterrupt(REMS_Int_Host_Port_0);
return (REMS_PL_OK);
}
```

This code block assigns and enables interrupts to all three lines with packets that are received on Queue 1 for the fido5100/fido5200 being separate from the other queues.

Table 3. PROFINET Specific Interrupt Signals

PROFINET Specific Interrupts	Indication	Usage/Handling
REMS_StdInt_TimerControl_0 = REMS_Int_Timer_Control_Event_0	Timer control unit (TCU) signal, send provider frame of AR 0	Handled by REMS driver
REMS_StdInt_TimerControl_1 = REMS_Int_Timer_Control_Event_1	TCU signal, send provider frame of AR 1	Handled by REMS driver
REMS_StdInt_TimerControl_2 = REMS_Int_Timer_Control_Event_2	TCU signal, send provider frame of AR 2	Handled by REMS driver
REMS_Int_Periodic_Timer_0	Provider data management during IRT legacy startup	Handled by REMS driver, IRT specific
REMS_Int_Periodic_Timer_1	Periodic processing of synchronization	Line delay thread (see the Synchronization (PTCP) section)
REMS_PnetInt_PktReady = REMS_Int_Queue_1_Packet_Ready	Nonreal-time traffic (NRT) packet received	Forward to PN stack and check for sync frame and sync events (see the Synchronization (PTCP) section)
REMS_PnetInt_CPM_Watchdog_Timeout0 = REMS_Int_Host_Port_0	AR 0 data Hold Timer (DHT) expired	Consumer scoreboard (CSB) handler (see the REMS_PnetInt_CPM_Watchdog_TimeoutX section)
REMS_PnetInt_CPM_Watchdog_Timeout1 = REMS_Int_Host_Port_1	AR 1 DHT expired	CSB handler (see the REMS_PnetInt_CPM_Watchdog_TimeoutX section)
REMS_PnetInt_CPM_Watchdog_Timeout2 = REMS_Int_Host_Port_2	AR 2 DHT expired	CSB handler (see the REMS_PnetInt_CPM_Watchdog_TimeoutX section)
REMS_PnetInt_CPM_Watchdog_Timeout3 = REMS_Int_Host_Port_3	AR 3 DHT expired	CSB handler (see the REMS_PnetInt_CPM_Watchdog_TimeoutX section)
REMS_PnetInt_ReceivedRtData0 = PROTOCOL_INT_ID	AR 0 cyclic packet received	CSB handler (see the REMS_PnetInt_ReceivedRtDataX section)
REMS_PnetInt_ReceivedRtData1	AR 1 cyclic packet received	CSB handler (see the REMS_PnetInt_ReceivedRtDataX section)
REMS_PnetInt_ReceivedRtData2	AR 2 cyclic packet received	CSB handler (see the REMS_PnetInt_ReceivedRtDataX section)

PROFINET

Table 3. PROFINET Specific Interrupt Signals (Continued)

PROFINET Specific Interrupts	Indication	Usage/Handling
REMS_PnetInt_ReceivedRtData3	AR 3 cyclic packet received	CSB handler (see the REMS_PnetInt_ReceivedRtDataX section)

Table 4. PROFINET IRT Interrupt Signals

PROFINET IRT Specific Interrupts	Indication	Usage/Handling
REMS_PnetInt_Leader_Lost		Internal use only
REMS_PnetInt_No_Sync_Message_Received	Synchronization lost, notify REMS driver and PROFINET stack	See the Synchronization (PTCP) section
REMS_PnetInt_Jitter_Out_Of_Boundary	Synchronization lost, notify REMS driver and PROFINET stack	See the Synchronization (PTCP) section
REMS_PnetInt_Sync	Synchronization achieved, notify REMS driver and PROFINET stack	See the Synchronization (PTCP) section
REMS_PnetInt_Wrong_Sync_Leader		Internal use only
REMS_PnetInt_PortStateRedUp		Internal use only

As the PROFINET REM switch driver supports multiple connections, there are factors to consider depending on the case. For single and multiconnection use cases, a syntax such as the following is needed to configure interrupts in general:

```
void REMS_PL_InterruptHandler(uint32_t line)
{
  volatile REMS_stdIntEvent_t event;
  REMS_CommonEnetPort_t port;
  interruptCnt_g++;
  REMS_StdEvaluateInterrupt((REMS_IntLine_t)line);

  do {
    event = REMS_StdGetNextEvent((REMS_IntLine_t)line);

    switch (event) {
    case ((REMS_stdIntEvent_t)0):
      break;
    /*...*/


    case REMS_StdInt_PktReady:
      ipFrameCnt_g++;
      LTE_SemSignal(REMS_PL_stdPacketSem_g);
      break;

    default:
      prtSpecificCnt_g++;
      REMS_PL_HandleIeEvent(line, event);
      break;
    }
  } while (event != (REMS_stdIntEvent_t)REMS_Int_None);

  return;
}
```

The implementation has REMS_PL_HandleEvent handling the PROFINET specific interrupts. This can be depending on the number of connections the user wants to support. If there is no system redundancy, then one connection is acceptable. Otherwise, handling of multiple

connections needs to be implemented. Refer to the Interrupts for One or Multiple Connections section for an exemplary description of the interrupt handling.

Interrupts for One or Multiple Connections

When implementing one connection, the user maps REMS_PL_HandleEvent(line,event)() such as the following:

```
void REMS_PL_HandleIeEvent(uint32_t line, REMS_stdIntEvent_t event)
{
  uint32_t arep;
  ETH_frameData_t *frameData_p;

  switch (event) {
  case ((REMS_stdIntEvent_t)0):
    break;

  case REMS_PnetInt_CPM_Watchdog_Timeout0:
    // for REMS_PnetInt_CPM_Watchdog_Timeout1 to -3 provide arep values 1 to
    arep = 0;
    // call EDDF_CRT_ISR_CSB_changed_IFA event number 12 - arep
    break;
  case REMS_PnetInt_ReceivedRtData0:
    // for REMS_PnetInt_ReceivedRtData1 to -3 provide arep values 1 to 3
    arep = 0;
    // call EDDF_CRT_ISR_CSB_changed_IFA event number 13 + arep
    break;
  case REMS_Int_Queue_1_Packet_Ready:
    LTE_SemSignal(REMS_PL_iePacketSem_g);
    break;
  default:
    break;
  }

  return;
}
```

Acyclic Message Reception

REMS Queue 1 is designated to receive PROFINET NRT frames including ACP, LLDP, PTCP, and MRP frames. After reading the packets from the driver, they must be provided to the PN stack. This is achieved by calling eddf_interrupt(pDDB->hDDB, EDDF_INT_ACYC_RCV_IFA). eddf_interrupt() leads to a call to EDDF_NRT_ISR_Acyc_Rcv_IFA(), which retrieves the current frame with a call to REMS_ReadPacket(). Also, make sure to clear the frame out of the REMS queue by a call to REMS_ReadPacket() in any error case.

```
sem_t q1PacketSem_g

// Signal Semaphore when receiving Q1 interrupt
void InterruptHandler(uint32_t line)
{
  volatile REMS_stdIntEvent_t event;
  REMS_CommonEnetPort_t port;
  REMS_StdEvaluateInterrupt((REMS_IntLine_t)line);

  do {
    event = REMS_StdGetNextEvent((REMS_IntLine_t)line);
```

```
    switch (event) {
    /*...*/
    /* Protocol-specific interrupt events */
    case REMS_Int_Queue_1_Packet_Ready:
      SignalSemaphore(q1PacketSem_g);
      break;

    /*...*/
    }
  } while (event != (REMS_stdIntEvent_t)REMS_Int_None);
  return;
}

void HandleQ1Packet(void *param)
{
  EDDF_LOCAL_DDB_PTR_TYPE    pDDB;

  do{
    LTE_SemTake(q1PacketSem_g, LTE_WAIT_FOREVER);
    pDDB = &g_EDDF_coreDev;
    if (pDDB->Core.Isr.Active) {
        eddf_interrupt(pDDB->hDDB, EDDF_INT_ACYC_RCV_IFA);
    } else {
      //read and dump packet
      REMS_ReadPacket(/*...*/);
    }
  } while (1);
}

/* called by PN-Stack*/
LSA_VOID EDDF_LOCAL_FCT_ATTR EDDF_NRT_ISR_Acyc_Rcv_IFA(
    EDDF_LOCAL_DDB_PTR_TYPE     pDDB,
    LSA_UINT32                  EventNr)
{
  /*...*/

  /* Allocate buffer for receive frame */

  /* Get NRT frame from Q1 */
  rv = REMS_ReadPacket(REMS_Queue_1,
                       &portNum,
                       &(ingressTime),
                            buf_p,
                       &size);
  frameHeader_p = (DCP_ETHDR*)buf_p;
  dataOffset = EDD_NRT_MIN_SND_LEN;
  //Get ethertype and skip VLAN if present
  /*...*/

  // Check ethertype and determine protocol
  // get User Channel RQB Queue
  switch (ethertype) {
    case 0x8892: //ACP_ETHERTYPE_RT:
      /*...*/
      break;
```

```
    case 0x88E3: //MRP_ETHERTYPE:
      /*...*/
      break;
    case 0x88CC: //LLDP_ETHERTYPE:
      /*...*/
      break;
    default:
      /* alert ip manager to presence of frame */
      isFramePass     = LSA_FALSE;
      ReleaseBuffer(buf_p);
      break;
  }

  // if protocol is known and RQB Queue is available, send RQB to Stack
  if (isFramePass)
  {
    if (pUsrChRqbQueue->Count)
    {
      /*...*/
    }
    else
    {
      ReleaseBuffer(buf_p);
    }
  }
}
```

Cyclic Data Exchange

The PN stack handles reception of cyclic data and their connected watchdog events separate from NRT traffic, via a CSB handler. The CSB handler provides the PN stack with information about the current status of cyclic traffic. This information is conveyed to the CSB handler via EDDF_CRT_ISR_CSB_changed_IFA().

REMS_PnetInt_CPM_Watchdog_TimeoutX

The REMS driver receives a timer interrupt (REMS_Int_Timer_Control_Int_X) by fido5100/fido5200. After the valid input of PROFINET configuration data, the driver monitors the reception of frames each send cycle. If no reception took place between two cycles, a miss counter is increased. When the miss count exceeds the configured watchdog counter (also called DHT), the driver issues REMS_PnetInt_CPM_Watchdog_TimeoutX. Check if the CSB at REMS driver has changed with REMS_HasCSBChanged() and if so, notify the PROFINET stack via eddf_interrupt(pDDB->hDDB, EDDF_INT_CSCOREBOARD_CHANGED_IFA).

REMS_PnetInt_ReceivedRtDataX

The REMS_PnetInt_ReceivedRtData event is published each time a new cyclic IO-frame is received, similar to REMS_PnetInt_CPM_Watchdog_TimeoutX. Each time this event occurs, call iom_swif_swap_cpm_buffer() to provide a new/free consumer buffer to REMS driver.

Check for a CSB change and notify the PROFINET stack of any changes via eddf_interrupt(pDDB->hDDB, EDDF_INT_CSCOREBOARD_CHANGED_IFA).

If the received data was provided by the primary AR (check with REMS_PnetIsArPrimary()), the data can be exchanged with the IO-application.

Isochronous Mode

To provide support for isochronous data exchange, the device must support record 0x8030 IsochronousModeData. The record must be stored for possible retrieval. To activate isochronous output signals, provide the received data to REMS driver via a call to REMS_PnetSetIsochronousParam().

After isochronous mode is active, T_i and T_o signals are issued on general-purpose inputs/outputs (GPIO) hardware pins, REMS_TCU_SIGNAL_0 and REMS_TCU_SIGNAL_1.

EDDF ADAPTION

The provided EDDF requires additional hardware-specific implementations. The EDDF Core General section, the EDDF NRT section, the EDDF PRM section, the EDDF PHY section, and the EDDF SYNC section provide information on the location and tasks of the required additions. Required changes are added as comments throughout the code snippets.

EDDF Core General

The following functions in **eddf_core_gen.c** require adaptions:

```
EDD_RSP EDDF_LOCAL_FCT_ATTR EDDF_GENLEDBlink(/*...*/)
{
  /*...*/
  // Interface HW function to blink PROFINET LEDs using EDDF_LED_BLINK_ON_OFF_DURATION_IN_100ms as cy▶
cle time

  EDDF_CORE_TRACE_01(pDDB->TraceIdx, LSA_TRACE_LEVEL_CHAT, "[H:%2X] OUT:
                     EDDF_GENLEDBlink()", pHDB->Handle)
  return (EDD_STS_OK);
}

LSA_VOID EDDF_LOCAL_FCT_ATTR EDDF_GENLEDBlink_Toggle(/*...*/)
{
  /*...*/
  if (pDDB->SWI.LEDBlink.TotalBlinkCountIn500ms <
(pDDB->SWI.LEDBlink.TotalBlinkDurationInSeconds * 2))
  {
    /* Toggle all LEDs of this interface to blink green*/
  }
  else
  {
    /*...*/
    /* Restore LEDs to initial state after blinking (continuously show green) */
  }
  /*...*/
}

LSA_VOID EDDF_LOCAL_FCT_ATTR _EDDF_GEN_ReadPhyStatusRegs(/*...*/)
{
  phyConfig_t phyConfig;

  EDDF_ASSERT_FALSE((LSA_HOST_PTR_ARE_EQUAL(pLinkStatus, LSA_NULL)));

  // Check for current LinkState of Port e.g. through REMS_GetPortLinkStatus()

  if (LinkStateUp(HWPortID-1)) == 0) {
    pLinkStatus->Link = EDD_LINK_DOWN;
    pLinkStatus->Speed = EDD_LINK_SPEED_100;
    pLinkStatus->Duplexity = EDD_LINK_MODE_FULL;
  } else {
    pLinkStatus->Link = EDD_LINK_UP;
    // Read Speed and Duplex config of Port
    GetPhyConfig(PNET_PL_ifHandle_g,
```

```
                  HWPortID-1),
                  &phyConfig,
                  sizeof(phyConfig));
    // Set pLinkStatus->Speed and pLinkStatus->Duplexity accordingly

  }
}
```

EDDF NRT

In **eddf_nrt_rcv.c**, return/release previously allocated buffers, if necessary as follows:

```
EDD_RSP EDDF_LOCAL_FCT_ATTR EDDF_NRTRecv(/*...*/)
{
  /*...*/
  /* Check if buffer has been allocated */
  if (pRQBRcv->UserDataLength > 0)
  {
    // Return/release pRQBRcv->pBuffer received to its origin, e.g., IP Stack
    pRQBRcv->UserDataLength = 0;
  }
  /*...*/
}
```

In **eddf_nrt_isr.c**, check for new cable/line delays and diagnosis events after evaluating a new PTCP message.

```
LSA_VOID EDDF_LOCAL_FCT_ATTR EDDF_NRT_ISR_Acyc_Rcv_IFA(/*…*/)
{
/*…*/
    switch (ethertype) {
        case 0x8892: //ACP_ETHERTYPE_RT:
          if ((frameID < 0x100) || ((frameID & 0xFF80) == 0xFF00)) {
            //PTCP
            /*…*/
            //Poll for diagnosis and delay events here

            OsFree(buf_p);
            return;
/*…*/
}
```

EDDF PRM

In **eddf_prm_inc.h**, provide the correct minimum frame send offset (FSO) time in nanosecond at define EDDF_PRM_PDIRFRAMEDATA_FRAMESENDOFFSET_MIN_EXAMPLE. The minimum FSO describes the minimum time after the start of an IRT red phase after which the device can inject a cyclic IO frame. If your device is not ready to send a frame after 1760 ns after the start of the red phase, adjust this value accordingly and also provide the changes in the general station description markup language (GSDML) file parameter MinFSO.

EDDF PHY

eddf_phy.c provides functions for the EDDF to get/set values of the underlying PHY. The following functions require adaptions to perform properly:

```
EDD_RSP EDDF_SYSTEM_OUT_FCT_ATTR EDDF_PHY_LoadDelayValues(/*...*/)
{
  int16_t rxDelay;
  int16_t txDelay;

  LSA_UNUSED_ARG(hDDB);

  // Read HW Phy delays from PHY and provide them in txDelay/rxDelay

  pLinkStatus->RealPortTxDelay    = txDelay;
  pLinkStatus->RealPortRxDelay    = rxDelay;
  pLinkStatus->MaxPortTxDelay     = txDelay;
  pLinkStatus->MaxPortRxDelay     = rxDelay;

  return EDD_STS_OK;
}

EDD_RSP EDDF_SYSTEM_OUT_FCT_ATTR EDDF_PHY_GetLinkStatus(/*...*/)
{
  /*...*/

  // ***** Examine passed parameters to determine link, speed, and duplexity.

  /* Depending on PHY speed provide values for:
   * pLinkStatus->Link
   * pLinkStatus->Speed
   * pLinkStatus->Duplexity
   * pLinkStatus->MAUType
   */

  pLinkStatus->AutonegCapAdvertised = 0;

  pLinkStatus->AutonegCapAdvertised |= EDD_AUTONEG_CAP_10BASET;

  /*...*/

return EDD_STS_OK;
}

EDD_RSP EDDF_SYSTEM_OUT_FCT_ATTR EDDF_PHY_SetPowerDown(/*...*/)
{
  LSA_UNUSED_ARG(hSysDev);

  if (EDDF_PHY_POWERDOWN == PowerDown)
  {
    // Power down phy
  }
  else
  {
    // Power up phy
}
```

```
  return EDD_STS_OK;
}

EDD_RSP EDDF_SYSTEM_OUT_FCT_ATTR EDDF_PHY_CheckPowerDown(/*...*/)
{
  LSA_UNUSED_ARG(hSysDev);

  /* Get PHY link status and set pIsPowerdown accordingly
   * Values: EDDF_PHY_POWERDOWN or EDDF_PHY_POWERUP
   */

  return EDD_STS_OK;
}

EDD_RSP EDDF_SYSTEM_OUT_FCT_ATTR EDDF_PHY_SetSpeedDuplexityOrAutoneg(/*...*/)
{
  /*...*/

  /* read current PHY config */
  if (EDD_AUTONEG_ON == Autoneg)
   {
    /* set PHY config to auto negotiation
     * speed/duplexmode/crossover values to unknown
     */
  }
  else
  {
    /* Set PHY config according to function parameters */
  }

  /*...*/
}
```

EDDF SYNC

In **eddf_sync.c**, retrieve receive and transmit PHY delays and report them to REMS driver. Additionally, provide function pointers to functions acquiring and releasing a mutex/binary semaphore as follows:

```
EDD_RSP EDDF_LOCAL_FCT_ATTR EDDF_SYNCDeviceSetup (/*…*/)
{
  int16_t rxDelay, txDelay;
  /* provide functions to acquire and release a mutex/binary semaphore*/
    void (*acquireMutex)(void) = NULL;
  void (*releaseMutex)(void) = NULL;
  /*…*/
  //GetPhyDelays(&rxDelay, &txDelay);
  REMS_PnetSetDelayValues(txDelay, rxDelay);
  REMS_SyncStartBridge(SYNC_SYNCID_CLOCK, acquireMutex, releaseMutex);
  /*…*/
}
```

ETHERNET/IP

The REM EtherNet/IP driver can be used to develop a device capable of supporting priority channel-based Ethernet/IP communications and, when combined with the DLR support library, beacon-based device level ring (DLR) redundancy. The REM switch is Open DeviceNet Vendors Association, Inc. (ODVA) conformant.

Additional capabilities of the REM switch include the following:

- Cut through operation
- IEEE-1588 end to end transparent clock
- Common industrial protocol (CIP) compliant quality of service (QOS) handling of EtherNet/IP Class 1 I/O frames (DSCP)

The creation of a complete EtherNet/IP device using the REM switch and this driver also requires a TCP/IP protocol stack and an EtherNet/IP protocol stack, both provided by the user. In addition, to support the DLR protocol details, combine this driver with the DLR support library. See the DLR support library user guide for more information on how to use the library.

The REM switch manages the Ethernet Layer 2 communications and switching, and also manages selected details of other protocol frames. Specifically, the REM switch detects EtherNet/IP Class 1 I/O frames that are directed to this device. Using one of the priority queues internal to the REM switch hardware, the REM switch redirects these frames to a higher priority queue. In addition, the REM switch detects DLR frames and redirects them to an independent queue. As such, there are three independent channels through which Ethernet frames can flow to the system software.

This version of the REM driver provides a static MAC address lookup table that contains space for six entries. This version of the REM driver for EtherNet/IP supports DLR as a DLR ring node only. This version does not support CIP synchronization.

The handling of DLR multicast frames does not require the use of the static table, regardless of whether the DLR is enabled.

ETHERNET/IP INITIALIZATION

To initialize the REM EtherNet/IP driver, complete the following steps:

1. Configure the external interrupt input pins on the host processor.
2. Call REMS_StdInit as follows:

```
REMS_StdInit(REMS_MII, 0, REMS_Int_Line_2, REMS_Int_Line_1, REMS_Int_Line_0);
```

   For the three priority levels, select the PHY mode (currently MII), 0 (the second argument is reserved, allowing for user specification), and the designation of the REM interrupt lines.
3. Optional. Call REMS_EipSetFilterCounters to provide the broadcast and multicast storm filter values. If this function is not called, the filters remain disabled.
4. Call REMS_StdSetMacAddress to communicate the system MAC address to the REM switch. This function takes three MAC addresses as parameters. When using EtherNet/IP, only one MAC address is necessary, and the second and third arguments can be zeros.
5. Call REMS_StdSetPortState once for each port to set the port state to REMS_PORT_FORWARDING.
6. Optional. Call REMS_EipSetDSCPValues to set the differential services code point (DSCP) QOS values. If this function is not called, the REM switch contains suitable default values for these settings and fully conforms to the ODVA requirements set forth in the CIP specification Volume 2 Section 5-7.4.2.

After completing these steps, the REM switch is fully configured and is ready to begin communications. There are two other considerations: handling the PHY link states and handling central processing unit (CPU) interrupts, which are discussed in the following sections.

HANDLING PHY LINK STATES

To function correctly, the REM switch must be told the external speed and duplex settings for each port. Because EtherNet/IP devices can be used in either 100 Mbps or 10 Mbps networks, do not assume a 100 Mbps full duplex. The REM driver provides a function named REMS_StdSetSpeedAndDuplex to change the speed and duplex settings. The REM medium priority interrupt events for port link change (REMS_Int_Port_1_Link_Change or REMS_Int_Port_2_Link_Change) must be used to trigger the process of reading the PHYs to determine the link speed and duplex, typically using the management data input/output (MDIO) serial management interface. After the medium priority interrupt events take place, call REMS_StdSetSpeedAndDuplex to update the switch settings.

ETHERNET/IP

The PHYs used with the REM switch are required to supply a link status output that can be routed to the REM switch input with the same name. This process triggers REMS_Int_Port_n_Link_Change. This PHY output must be the link status, not link activity or status. A typical PHY has a low active output (generally intended to drive a light emitting diode (LED)) that is often set by default to also act as an activity indicator. If unchanged, this output continually toggles as communications proceed, triggering erroneous link up and link down interrupts. The PHY configuration must be changed so that this toggling does not occur.

The REM switch and driver do not provide any MDIO hardware or subroutines. The system designer must supply these subroutines.

HANDLING CPU INTERRUPTS

During the REM initialization process, calling REMS_StdInit designates the REM interrupt lines high, medium, or low priority. The REM switch and driver then assign these interrupt lines to the various interrupt events as described in Table 5.

Table 5. Interrupt Priorities

Priority	Event	Purpose
High	REMS_Int_Queue_1_Packet_Ready	EtherNet/IP Class 1 frame received
Medium	REMS_Int_Port_1_Link_Change	Port 1 link up or down detect
Medium	REMS_Int_Port_2_Link_Change	Port 2 link up or down detect
Medium	REMS_Int_Port_1_0	Port 1 unintended loop detected
Medium	REMS_Int_Port_1_1	Port 1 DLR beacon timeout
Medium	REMS_Int_Port_2_0	Port 2 unintended loop detected
Medium	REMS_Int_Port_2_1	Port 2 DLR beacon timeout
Low	REMS_Int_Queue_0_Packet_Ready	TCP/IP frame received
Low	REMS_Int_Queue_2_Packet_Ready	DLR frame received

The user determines how to set up the host low level interrupt request line (IRQ) handler. After being set up, the IRQ handler can call a REM event handler function. The IRQ handler must have access to the REM interrupt line being used and may use various REM driver functions to perform the following sequence:

1. At the start of the handler, call REMS_StdEvaluateInterrupt one time only to obtain a complete list of all events currently pending for the interrupt line of interest. This function not only retrieves this event list but also acknowledges all the pending and enabled interrupt events detected within the REM hardware. All detected events must be handled in the same interrupt.
2. In a while loop, repeatedly call REMS_StdGetNextEvent to get the next pending event until the event returned is REMS_Int_None. For each event retrieved, call the appropriate code to handle that event. This process is described in detail in the Low Priority TCP/IP Frame Receive and Transmit Processing, High Priority EtherNet/IP Class 1 Frame Receive and Transmit Processing, and EtherNet/IP DLR Frame Receive and Transmit Processing sections.

See the Interrupts section for general examples of this process. For EtherNet/IP, the user must handle interrupt events.

LOW PRIORITY TCP/IP FRAME RECEIVE AND TRANSMIT PROCESSING

The setup and utilization of the TCP/IP stack is up to the user. This user guide describes how to use the REM driver to connect the REM switch hardware to the stack using the following three mechanisms, which the user stack must contain:

- A buffer pool for placing received frames.
- A mechanism for notifying if frames have been received.
- A way to register a callback function that the stack uses to transmit a frame when desired.

Receive

When a low priority frame arrives, the REM hardware triggers the low priority interrupt line and issues a REMS_Int_Queue_0_Packet_Ready event. The user handler must be ready to detect this event, as well as perform the following sequence:

1. Get a buffer from the buffer pool of the TCP/IP stack.
2. Use the REMS_StdReadPacket or REMS_StdReadPacketWithTimestamp REM driver functions to read the data from the switch queue into that buffer. Note that the difference between these two functions involves the return of the ingress timestamp, which is the time at which the packet was received.
3. Use the notification mechanism of the TCP/IP stack to tell the TCP/IP stack a received frame is ready.

Transmit

To transmit a low priority TCP/IP frame, no REM interrupt or event handler is used. Assuming a transmit handler function can be registered with the TCP/IP stack, the user must only write this function and register the function with the stack. The function must perform the following sequence:

1. Get the frame from the TCP/IP stack.
2. Call either REMS_StdXmitTaggedPacket, REMS_StdXmitPacket, or REMS_StdXmitPacketWithControlFlag, as required to send data through the switch queue. The difference between the first two functions involves the use of an IEEE-802.1Q VLAN tag. Many TCP/IP stacks do not have the capability of tagging the frames themselves. If this is the case, use the tagged version and supply the tag data to the REM driver separately. The driver inserts the tag at the location designated. The third function allows the command of an egress timestamp insertion or capture. If an egress timestamp insertion is called, the hardware overwrites packet data at the location with the current timestamp as the packet is transmitted. Likewise, if a timestamp capture is called, the hardware captures the timestamp when the packet is transmitted. The commands and insertion sizes used to create the control flags are defined in **REMS_Basic.h**.

HIGH PRIORITY ETHERNET/IP CLASS 1 FRAME RECEIVE AND TRANSMIT PROCESSING

The REM hardware detects EtherNet/IP Class 1 I/O frames and directs them to a dedicated high priority receive queue. To ensure undisturbed handling of this data, this queue is used for no other purpose. Because the frames used to set up the EtherNet/IP Class 1 connection (such as register session and forward open) are not Class 1 I/O frames, those frames are not directed to the high priority queue. This communication takes place in a low priority queue.

If the user EtherNet/IP stack can make use of the independent flow of this data, connecting EtherNet/IP stack to this queue to create a high priority I/O channel is simple. If the user stack does not have this capability, follow the steps described in the Low Priority TCP/IP Frame Receive and Transmit Processing section.

Receive

When a high priority Class 1 I/O frame arrives, the REM hardware triggers the high priority interrupt line and issues a REMS_Int_Queue_1_Packet_Ready event. The user handler must be able to detect this event, as well as perform the following sequence:

1. Get a buffer from the buffer pool of the EtherNet/IP stack.
2. Use the REMS_Class1ReadPacket REM driver function to read the data from the switch queue into that buffer.
3. Use the notification mechanism of the EtherNet/IP stack to inform the EtherNet/IP stack that a received frame is ready.

Transmit

To transmit a high priority Class 1 I/O frame, use of a REM interrupt or event handler is not necessary. Assuming a transmit handler function can be registered with the EtherNet/IP stack, the user must only write this function and register it. The function must perform the following sequence:

1. Retrieve the frame from the EtherNet/IP stack.
2. Call either REMS_Class1XmitTaggedPacket or REMS_Class1XmitPacket to send data through the switch queue. The difference between these two functions involves the use of an IEEE-802.1Q VLAN tag. Many TCP/IP stacks do not have the capability of tagging the frames themselves. If this is the case, use the tagged version and supply the tag data to the REM driver separately. The driver inserts the tag at the location designated.

ETHERNET/IP DLR FRAME RECEIVE AND TRANSMIT PROCESSING

The REM hardware detects EtherNet/IP DLR frames and directs these frames to a dedicated receive queue. To ensure the undisturbed handling of this data, this queue is used for no other purpose.

Receive

When an EtherNet/IP DLR frame arrives, the REM hardware triggers the low priority interrupt line and issues a REMS_Int_Queue_2_Packet_Ready event. The user handler must be able to detect this event, as well as perform the following tasks:

1. Get a buffer from the buffer pool of the EtherNet/IP DLR stack.
2. Use the REMS_DlrReadPacket REM driver function to read the data from the switch queue into that buffer.

3. Use the notification mechanism of the EtherNet/IP DLR stack to inform the EtherNet/IP DLR stack that a received frame is ready.

The DLR support library uses a software queuing mechanism to receive these frames. This queue is created by the board support package files and is referred to by the g_DLR_PacketQueue variable name. The frames, when received, are added to this queue and are retrieved when the DLR library calls the BSP_Get_DLR_Packet board support package function. This process takes place when the RING_EVENT_RECEIVE_MSG event is processed within the EtherIpRingProtocol_ProcessEvents function.

Transmit

To transmit an EtherNet/IP DLR frame, no REM interrupt or event handler is used. Assuming a transmit handler function can be registered with the Ethernet/IP DLR stack, the user must only write this function and register it. The function must perform the following sequence:

1. Get the frame from the EtherNet/IP DLR stack
2. Call REMS_DlrXmitPacket to send data through the switch queue.

The DLR support library has a function to transmit DLR frames when necessary. The function is integrated in the DLR library and, in this case, does not need to be registered. The function is called BSP_Put_DLR_Packet and can be found in the DLR library board support package file, **BspEnetSwitch.c**.

OTHER CONSIDERATIONS FOR DLR

By default, on power up, the REM switch disables all DLR features. It is necessary to specifically enable REM switch DLR features by using the REM driver function REMS_DlrEnable.

It is also necessary to use the DLR support library to implement the software details of the DLR protocol itself. The initialization of the support library is described in the DLR support library user guide. After the REM switch or driver and the DLR library are initialized, route the received DLR frames to the DLR support library.

DLR Frame Handling by Frame Type

When the DLR is enabled, all DLR frames are routed by the REM switch. There is no need to use an entry in the static routing table. The specific routing of the DLR frames is described in Table 6.

Table 6. DLR Frame Routing

Destination MAC	Frame	Forward Port to Port	Forward to Host
01-21-6C-00-00-01	Beacon	Always	Conditional
01-21-6C-00-00-02	Neighbor_Check_Request	Never	Always
01-21-6C-00-00-02	Neighbor_Check_Response	Never	Always
01-21-6C-00-00-02	Sign On	Never	Always
01-21-6C-00-00-03	Announce	Always	Always
01-21-6C-00-00-03	Locate_Fault	Always	Always
01-21-6C-00-00-03	Flush_Tables	Always	Always
01-21-6C-00-00-04	Advertise	Always	Always
01-21-6C-00-00-05	Learning Update	Always	Always
MAC of Active Super	Link_Status	Always	Always

Handling DLR Beacon Frames

The REM switch sends beacon frames to the host only if the DLR ring state changes or if the active DLR supervisor MAC changes. When one of these events occurs, expect one frame from each port. When the beacon frames are received and successfully processed by the user code or DLR stack, extract the beacon interval and timeout data and use them to program the TCU by using the REMS_EipStartTcu REM driver function. After the TCU is started, call REMS_EipEnableBTOIrq to enable the beacon timeout interrupt. This interrupt is routed to the medium priority interrupt and the REM events. REMS_Int_Port_1_1 and REMS_Int_Port_2_1 are used to indicate this interrupt. Each port independently monitors for beacon timeouts.

Successful processing of the beacon frames means that a valid beacon frame has been received from both ports, and both frames indicate that the system has made the transition into ring normal state. Only start the TCU once. At this point, the REM switch begins to process received beacon frames to monitor for timeouts, active supervisor MAC changes, or ring state changes.

ETHERNET/IP

Handling Unintended Loop Detect

The external ports on the REM switch examine the source MAC of each received frame to detect frames with a source MAC that matches that of the system. If the external ports detect a matching frame, an Ethernet loop exists. These frames are not forwarded to the other port, but instead generate a medium priority interrupt to the host. The interrupt events generated are REMS_Int_Port_1_0 and REMS_Int_Port_2_0. These events allow the user DLR stack to report the detection of an unintentional loop.

Handling Port Link Change

When an Ethernet link up or link down event occurs, the DLR library must be informed. The event itself is an interrupt that the REM switch generates to the host CPU (see Table 5). The EtherIpRingProtocol_HandleLinkStateChange DLR library function is called in response. See the DLR library user guide for more information.

BROADCAST AND MULTICAST FILTERING

When using the EtherNet/IP protocol, the REM switch can be set to limit the rate of broadcast and multicast frames that are routed through the switch. This setup is referred to as broadcast or multicast storm protection. This protection is implemented with an adjustable threshold that allows the REM switch to only accept n number of frames in t milliseconds.

To set up this protection, the REM switch counts the broadcast and multicast frames accepted until it reaches n and then begins to discard them. Each frame type (broadcast or multicast) is counted separately so there are two independent filters. The host CPU can enable this filtering by calling REMS_EipSetFilterCounters to set the packet limits. There is no specific mechanism to set the time period, and it is only necessary to periodically signal the switch to reset the limit counters. The REM switch driver function that resets the limit counters is named REMS_EipServiceBcastMcastFilter. Because this signal resets both counters, there is only one time interval shared between both filters. Although it is possible to use one of the periodic timers of the REM switch for this purpose, the creation and maintenance of such a function is user dependent.

MODBUS/TCP

This REM switch driver software provides a mechanism that allows the exchange of I/O data information and Modbus/TCP configuration with the REM switch. Use this software driver with the REM switch to implement a priority channel in a Modbus/TCP device.

To complete a Modbus/TCP device, a TCP/IP protocol stack and a Modbus/TCP follower stack is required. These stacks are not provided as part of this software package. This driver package handles communication between the application and stacks and the REM switch.

The REM switch ensures Layer 2 switch functionality (such as broadcast and multicast frame routing, static table, and dynamic table) and also prioritizes Modbus/TCP traffic above all other traffic. The REM switch expects packets from the driver to be written to different host write queues depending on their priority. Packets written to the high priority queue are transmitted before any packets that are written to the standard priority queue. In this way, the application can write standard, nonModbus/TCP packets to the standard priority queue and can write Modbus/TCP packets to the high priority queue to ensure that Modbus/TCP packets are transmitted with preference over standard packets.

Similarly, the REM switch examines all received unicast frames intended for the device to determine if they are Modbus/TCP frames. If the packet is determined to be a Modbus/TCP packet, the packet is routed to the high priority host read queue. All other traffic is routed to the standard priority host read queue. As such, the attached host processor can give preference to Modbus/TCP packets over all other packets, ensuring that Modbus/TCP packets always arrive at the host processor, regardless of network loading.

MODBUS/TCP INITIALIZATION

The software initialization procedure of the REM switch and driver for Modbus/TCP is as follows:

1. Configure the external interrupt input pins on the host processor.
2. Assert the REM switch reset line and wait for the switch to become ready.
3. Call REMS_StdInit(). Provide the PHY mode (currently MII), the clock enable flag (set to 0), and the REM switch interrupt lines you want to use for the three priority levels.
4. Call REMS_StdSetMacAddress() to set the MAC address for the system. There are three MAC address parameters to this function. For Modbus/TCP, supply the same MAC address for each of the three MAC address parameters.

MODBUS/TCP INTERRUPT HANDLING

When REMS_StdInit() is called, the interrupt outputs are assigned high, medium, and low priority positions according to the passed in parameters. The REM switch driver assigns interrupt sources to interrupt lines, as shown in Table 7. The low-level interrupt handler must be aware of which REM switch interrupt line caused the interrupt and then call the REM switch event handler function (as shown in the Interrupts section). After the event handler function is called, all pending interrupts are evaluated and handled individually in a loop.

Table 7. Interrupt Sources for Modbus/TCP

Priority	Event	Purpose
Low	REMS_StdInt_Port_1_LinkChange	Port 1 link up/down
Low	REMS_StdInt_Port_1_LinkChange	Port 2 link up/down
Low	REMS_Int_Queue_1_Packet_Ready	Standard priority packet received
High	REMS_Int_Queue_0_Packet_Ready	Modbus/TCP packet received

MODBUS/TCP PHY LINK STATE INTERRUPT HANDLING

The REM switch determines link up or down from a signal supplied to it by the PHYs, but the link speed and duplex settings must be written to the REM switch by the host processor. The REM switch does not determine these settings on its own. When the EtherNet link becomes active (REMS_StdInt_Port_1_LinkChange or REMS_StdInt_Port_2_LinkChange interrupt event occurs), the application reads the link speed and duplex from the PHYs. The link speed and duplex are then written to the REM switch using REMS_StdSetSpeedAndDuplex() to keep the REM switch updated with the current link settings. The REM switch and driver do not provide MDIO hardware or driver subroutines. The system designer is responsible for providing a mechanism to deliver link and duplex information to the REM switch.

MODBUS/TCP RECEIVED PACKET INTERRUPT HANDLING

When the REM switch triggers the REMS_Int_Queue_0_Packet_Ready or REMS_Int_Queue_1_Packet_Ready event, a high or standard priority packet for the device has arrived and is ready for the host processor to read from the REM switch memory. The interrupt handler then supplies the new packet to the TCP/IP stack. Depending on the architecture of the TCP/IP stack to which the packet is supplied, the process of supplying the stack may appear as follows:

1. Check for a free buffer from the TCP/IP stack.

2. If a free buffer is found, use REMS_StdReadPacket() for standard priority packets or REMS_Read_ModbusTCP_Packet() for Modbus/TCP packets to read the new packet from the REM switch memory into the buffer.
3. Supply the buffer to the TCP/IP stack to begin processing.

All received TCP/IP packets with a source or destination port number of 502 generate a REMS_Int_Queue_0_Packet_Ready interrupt event. All other received packets generate a REMS_Int_Queue_1_Packet_Ready interrupt event. Because the REMS_Int_Queue_0_Packet_Ready interrupt event has a higher priority than the REMS_Int_Queue_1_Packet_Ready interrupt event, Modbus/TCP packets inherently arrive at the host processor at a higher priority than other packets.

MODBUS/TCP PACKET TRANSMISSION

Transmission of frames can be initiated at any time. The system designer registers a packet transmission routine with the TCP/IP stack to connect the TCP/IP stack to the Ethernet driver. This process is also true when using the REM switch. When it is time to transmit a packet, the TCP/IP stack calls the packet transmission routine to retrieve the packet from the TCP/IP stack and supply the packet to the REM switch for transmission using REMS_StdXmitPacket()or REMS_Xmit_ModbusTCP_Packet(). REMS_StdXmitPacket() writes the packet to the REM switch using the standard priority queue while REMS_Xmit_ModbusTCP_Packet() writes the packet to the REM switch using the high priority queue. In this way, Modbus/TCP packets transmit from the device at a higher priority than standard packets.

ETHERCAT

The EtherCAT driver provides a software interface by which an application layer and EtherCAT follower stack can initialize and use the REM switch as an EtherCAT follower controller (ESC). When this driver is used, the Beckhoff EtherCAT follower stack code (SSC) can be integrated with minimal porting effort because the driver is designed to interact directly with the SSC.

To create a finished EtherCAT device, an EtherCAT follower stack must be supplied by the user. The driver is designed to directly integrate with the Beckhoff EtherCAT SSC. The REM switch, in combination with this driver, acts as an ESC that supports distributed clocks, eight synchronization managers (hereafter referred to as SyncManager), and eight Fieldbus Memory Management Units (FMMUs) with 10 kB of RAM.

When a user is going to create an EtherCAT follower device using the fido5200, there is a factor to consider. EtherCAT has two ports: one is an inbound port and the other is the outbound port. When designing a board, all of the terminologies used for the port numbers start with 0 as opposed to other protocols starting with 1. Therefore, Port 0 is the inbound port (also called Port 1 for other protocols) and Port 1 is the outbound port (also known as Port 2 in other protocols).

ETHERCAT INITIALIZATION

To initialize an EtherCAT follower stack, perform the following actions:

1. Disable PHYs and power down both PHYs via MDIO to prevent any network traffic from entering until the REM switch and the system are completely initialized and ready for EtherCAT communication.
2. Restore electrically erasable programmable read only memory (EEPROM) emulation data. Retrieve the emulated EEPROM data from nonvolatile memory. The REM switch does not have attached EEPROM so the EEPROM must be emulated by the application. This requirement means that the application must provide functions for the EtherCAT leader to read and write EEPROM. See the EEPROM Emulation section for more details.
3. Reset the REM switch. This reset ensures proper system startup and that the host processor software and the REM switch firmware are synchronized. Initialize the REM switch. Perform the standard REM switch initialization by calling REMS_StdInit(). Use MII for the PHY mode and leave the clock enabled. A call to REMS_StdInit() may appear as follows:

   ```
   REMS_StdInit(REMS_MII, 0, REMS_Int_Line_2, REMS_Int_Line_1, REMS_Int_Line_0);
   ```

   Calling REM_StdInit() in this way is highly recommended for EtherCAT as it maps the priorities per what is expected for the EtherCAT version of the REMS Driver.
4. Set the synchronization offset value. Call REMS_ecatSetSyncOffsetValue() to inform the driver of the delay across the PHYs. This information is used to compensate for the propagation delay across the PHYs, because this information relates to maintaining synchronization using distributed clocks. No other functions declared in **REMS_ECATinternals.h** need to be called by the application. This driver assumes that both ports have identical PHYs and thus that the PHY delay values are the same.
5. Enable PHYs. Power up both PHYs via MII. Now that the REM switch is ready for network communication, the PHYs can be enabled, which allows packets to reach the REMS.
6. Initialize the EtherCAT follower stack. If the Beckhoff EtherCAT SSC is being used, call MainInit(). Otherwise, initialize the EtherCAT follower stack appropriate for the user system.
7. Start the EtherCAT follower stack main processing. If the Beckhoff EtherCAT SSC is being used, call MainLoop() according to the conditions provided in the associated documentation from Beckhoff Automation. Otherwise, start the EtherCATfollower stack that is in use.

ETHERCAT INTERRUPT HANDLING

The REM switch provides the ability to have three prioritized interrupt lines.

The REM switch chip and this driver communicate frequently. The driver reads and writes to the REM switch and the REM switch requests interrupts to the driver for reads, writes, and other actions. Not all of these interrupts must be handled by the application. Most are handled internally by the driver. The remainder of the interrupts must be handled by the application, as shown in Table 8.

Table 8. Interrupt Handling for EtherCAT

Priority	Event	Purpose
Low	REMS_StdInt_Port_1_LinkChange	Port 1 link up and down detect
Low	REMS_StdInt_Port_2_LinkChange	Port 2 link up and down detect
Low	REMS_EcatInt_MII_MGT_Event	New media independent interface management interface (MIIMI) command to be handled
Low	REMS_EcatInt_Reset_Requested	The EtherCAT leader requests the device to reset

ETHERCAT

Table 8. Interrupt Handling for EtherCAT (Continued)

Priority	Event	Purpose
Low	REMS_EcatInt_AL_Event_Change	Equivalent to process data interface (PDI) interrupt
High	REMS_EcatInt_SYNC0_Event	Equivalent to SYNC0 interrupt
High	REMS_EcatInt_SYNC1_Event	Equivalent to SYNC1 interrupt

A recommended function structure for handling interrupts from the REM switch follows. This example shows interrupts being internally handled by the driver and interrupts that are supplied to the application for handling. This function is intended to be called directly from the host processor interrupt service routine.

```
void HandleIntREMS(REMS_IntLine_t line)
{
        REMS_stdIntEvent_t event;

        REMS_StdEvaluateInterrupt(line);
        do {
                // Interrupts for the driver are evaluated and handled here
                event = REMS_StdGetNextEvent(line);

                // Interrupts for the application are handled here
                switch (event) {
                case REMS_StdInt_Port_1_LinkChange:
                case REMS_StdInt_Port_2_LinkChange:
                        // Call link change handler here
                        break;

                case REMS_EcatInt_AL_Event_Change:
                        PDI_Isr();      // Call stack handler
                        break;

                case REMS_EcatInt_SYNC0_Event:
                        Sync0_Isr();    // Call stack handler
                        break;

                case REMS_EcatInt_SYNC1_Event:
                        Sync1_Isr();    // Call stack handler
                        break;

                case REMS_EcatInt_MII_MGT_Event:
                        // Signal external process to handle MIIMI command here
                        break;

                case REMS_EcatInt_Reset_Requested:
                        // Reset host processor
                        break;

                default:
                        break;
                }
        } while (event != REMS_Int_None);
}
```

The driver requires the host processor to be configured for level sensitive interrupts on the interrupt lines. The application waits to acknowledge the actual interrupt until after HandleIntREMS() returns. If additional REM switch interrupts are generated while the current interrupts are being

processed, the external CPU interrupt line remains high. When the CPU interrupt is acknowledged, a new CPU interrupt is requested, which ensures that all interrupt requests are processed.

ETHERCAT FOLLOWER STACK TO DRIVER INTERFACE

Although this driver can be used with any stack, the driver is designed to work with Version 5.11 and Version 5.12 of the EtherCAT follower stack from Beckhoff Automation. All porting layer functions that the EtherCAT follower stack expects, such as HW_EscRead(), are defined in this driver.

When using the EtherCAT SSC tool, the HW_ACCESS_FILE configuration parameter allows the user to specify the header file that contains the low-level hardware access functions. Use this parameter in the SSC tool to specify the **REMS_ECATHw.h** file in this driver. The value for the HW_ACCESS_FILE parameter in the SSC tool may appear as follows:

```
#include "inc/REMS_ECATHw.h"
```

Included in the driver package is a sample SSC tool settings file called **SSC_Tool_Settings.esp**. Open this file with the SSC tool for an example of the settings that can generate the EtherCAT SSC.

EEPROM Emulation

The host processor is responsible for emulating the EEPROM that typically connects directly to the ESC. This emulation includes handling EEPROM read commands, write commands, and reload commands. A reload command is issued at system startup. The EtherCAT leader commands the reload command at any time. While servicing the reload command, certain data from EEPROM must be placed in the EEPROM data register in packed form.

MII MANAGEMENT INTERFACE

Most ESCs can access registers on the connected PHYs via the MDIO and MDC lines. The REM switch does not have these signals, and thus does not have the ability to directly read or write registers on connected PHYs.

ESC Register 0x0510 to Register 0x0515 allow the EtherCAT leader to command the ESC to access the registers on the connected PHYs. Because the REM switch does not have the MDIO and MDC signals, the REM switch cannot fulfill the PHY register read or write requests. To accomplish the PHY register read or write, the REM switch requests that the host processor perform the PHY register read or write in its place.

When an EtherCAT leader generates a PHY register read or write, the REM switch generates a REMS_EcatInt_MII_MGT_Event event. This event signals the host processor to perform the PHY register read or write in the place of the ESC, and then provides the results of the operation. The application layer software must perform the action and indicate if the operation is successful or failed, as well as the result of the operation if successful. In this process, the MIIMI is emulated.

Similarly, ESC Register 0x0301 and Register 0x0303 contain the number of receive errors that occurred. The REM switch does not have an RX_ERR pin to connect the RX_ERR signal from the PHYs. To report receive errors, the REM switch must periodically request the host processor to read the receive error register on the PHYs to keep Register 0x0301 and Register 0x0303 updated. The REM switch accomplishes this read through the MIIMI emulation mechanism. The REM switch generates a REMS_EcatInt_MII_MGT_Event event to begin the read of the receive error register on the indicated PHY.

When the EtherCAT leader issues an MIIMI request, or when the REM switch attempts to periodically read a receive error register from a connected PHY, a REMS_EcatInt_MII_MGT_Event event is generated. When this event is generated, the application calls REMS_ecatMiiEventParams(), declared in **REMS_ECATHw.h**, to obtain the parameters for the MIIMI command. After the command parameters are obtained, the command can be executed.

When the application responds to a REMS_EcatInt_MII_MGT_Event event, the application may be responding to one of two functions. To determine whether the application is responding to a PHY register read or write from the EtherCAT leader or to a periodic read of a PHY receive error register, examine the values in the variables populated by REMS_ecatMiiEventParams() after the application returns. The suggested process for responding to a REMS_EcatInt_MII_MGT_Event event is as follows:

1. Call REMS_ecatMiiEventParams() to retrieve the parameters for the REMS_EcatInt_MII_MGT_Event event.
2. Map the indicated PHY address to the actual PHY address. The EtherCAT leader assumes that the PHY connected to Port 0 has Address 0 on the MDIO bus, and that the PHY connected to Port 1 has Address 1 on the MDIO bus. If this is not the case, remap the PHY addresses

so that the intended PHY is accessed. The REM switch makes this same assumption when issuing periodic reads of the receive error registers.

3. Optional. Verify that the PHY register that is to be read or written is supported on the indicated PHY. Set a local error variable if not.
   a. If the REM switch is requesting the receive error register value from PHY 0, read the receive error register on the PHY attached to Port 0. Call REMS_ecatMiiReadComplete() and indicate if any errors occurred, the value of the receive error register if no errors occurred, and that the periodic receive error register read is complete (why parameter = 0).
   b. If the REM switch requests the receive error register value from PHY 1, read the receive error register on the PHY attached to Port 1. Call REMS_ecatMiiReadComplete() and indicate if any errors occurred, the value of the receive error register if no errors occurred, and that the periodic receive error register read is complete (why parameter = 0).
   c. If the EtherCAT leader requests a PHY register read, read the indicated register address from the PHY at the remapped MDIO bus address. Call REMS_ecatMiiReadComplete() and indicate if any errors occurred, the value requested from the indicated register on the indicated PHY if no errors occurred, and that the standard PHY register read is complete (why parameter = 1).
   d. If the EtherCAT leader requests a PHY register write, write the indicated PHY register value to the indicated PHY register at the remapped MDIO bus address. Call REMS_ecatMiiWriteComplete() and indicate if any errors occurred.

Because the transaction on the MDIO/MDC lines can take a relatively large amount of time, avoid performing the PHY read or write inside the interrupt service routine that handles the REMS_EcatInt_MII_MGT_Event event.

It is possible to receive a REMS_EcatInt_MII_MGT_Event event while REMS_EcatInt_AL_Event_Change, REMS_EcatInt_SYNC0_Event, and REMS_EcatInt_SYNC1_Event are disabled. Because of this ability, it is recommended to ensure that a REMS_EcatInt_MII_ MGT_Event event is not serviced while the other events are disabled. The relatively large amount of time the MDIO transaction takes may prevent other events from being handled quickly enough during standard operation.

ETHERCAT SSC

The EtherCAT SSC driver operates with the EtherCAT follower stack from Beckhoff Automation. There is no need to review this section if using a different follower stack.

SSC Tool Settings

The SSC tool settings file used to create the SSC for the RapID platform is included in this package. The user can modify this existing file or create a new settings file. Some settings in the SSC tool are restricted to maintain compatibility with this driver and ease integration. These settings are described in Table 9.

Table 9. SSC Tool Restricted Settings

Parameter	Restriction	Description
Generic		
EXPLICIT_DEVICE_ID	0 (fixed)	Explicit device identification is not supported.
ESC_SM_WD_SUPPORTED	0 (fixed)	Using the SyncManager watchdog on the REM switch is a poor choice due to PDI type.
ESC_EEPROM_ACCESS_SUPPORT	0 (fixed)	The host processor emulates EEPROM, therefore no access functions must be provided. Host processor can access the emulated EEPROM values directly.
Hardware		
EL9800_HW	0 (fixed)	EL9800 hardware is not used.
MCI_HW	0 (fixed)	The stack does not expect a MCI to the ESC.
FC1100_HW	0 (fixed)	FC1100 hardware is not used.
_PIC18	0 (fixed)	EL9800 hardware is not used.
_PIC24	0 (fixed)	EL9800 hardware is not used.
EXT_DEBUGGER_INTERFACE	0 (fixed)	EL9800 hardware is not used.
UC_SET_ECAT_LED	1 (fixed)	Host µC must provide run and error LEDs.
ESC_SUPPORT_ECAT_LED	0 (fixed)	REM switch does not provide support for run and error LEDs.
ESC_EEPROM_EMULATION	1 (fixed)	Host processor must emulate required EEPROM.
EEPROM_READ_SIZE	4 (fixed)	REM switch EEPROM data register size is fixed at 4 bytes.
ESC_CONFIG_DATA	0x08040066204E (fixed)	Ensures information reloaded from EEPROM on boot is correct.

Table 9. SSC Tool Restricted Settings (Continued)

Parameter	Restriction	Description
MAKE_PTR_TO_ESC	Empty (fixed)	ESC is not interfaced as memory.
EtherCAT State Machine		
BOOTSTRAPMODE_SUPPORTED	0 (fixed)	REM switch does not support bootstrap mode.
Application		
TEST_APPLICATION	0 (fixed)	Do not use this setting to create a real application.
EL9800_APPLICATION	0 (fixed)	EL9800 hardware is not used.
SAMPLE_APPLICATION	0 (fixed)	Does not use the sample application.
SAMPLE_APPLICATION_INTERFACE	0 (fixed)	Does not use the sample application.
Application.ProcessData[1]		
MIN_PD_WRITE_ADDRESS	Not applicable	Not applicable
DEF_PD_WRITE_ADDRESS	Not applicable	Not applicable
MAX_PD_WRITE_ADDRESS	Not applicable	Not applicable
MIN_PD_READ_ADDRESS	Not applicable	Not applicable
DEF_PD_READ_ADDRESS	Not applicable	Not applicable
MAX_PD_READ_ADDRESS	Not applicable	Not applicable
MAX_PD_INPUT_SIZE	Not applicable	Not applicable
MAX_PD_OUTPUT_SIZE	Not applicable	Not applicable
Mailbox[1]		
MIN_MBX_WRITE_SIZE	Not applicable	Not applicable
DEF_MBX_WRITE_SIZE	Not applicable	Not applicable
MAX_MBX_WRITE_SIZE	Not applicable	Not applicable
MIN_MBX_READ_SIZE	Not applicable	Not applicable
DEF_MBX_READ_SIZE	Not applicable	Not applicable
MAX_MBX_READ_SIZE	Not applicable	Not applicable

[1] When selecting minimum or maximum addresses for process data reads and writes, maximum process data input/output sizes and minimum, default, and maximum addresses for mailboxes, bear in mind that the REM switch has 10 kB of RAM that can be used. This space is available to accommodate space for the input mailbox, output mailbox, triple buffered input process data, and triple buffered output process data. Pick all of the minimum, maximum, and default values so that there is enough RAM available for all operations. The SSC tool settings file sets the ALLOCMEM (size), FREEMEM (pointer), APPL_AllocMailboxBuffer (size), and APPL_FreeMailboxBuff (pointer) parameters to application specific functions. Unless the system requires something different, malloc() and free() can be used for these parameters.

SSC Changes

To obtain correctly functioning code, minor changes are made to the SSC after the SSC tool produced the source code. The majority of these changes relate to the SSC not running correctly on a big endian host processor. These changes are detailed in this section. The user may or may not need to make the same changes when configuring the SSC tool, and the following changes are provided as a reference material. If the user needs to make these changes, the actual line numbers may vary. The following line numbers are used for Version 5.12 of the EtherCAT stack. If you are using Version 5.13 or onward, these line numbers may vary.

In **ecatappl.c**, lines 889 to 891, added:

```
else
{{
EEPROMReg &= ~ESC_EEPROM_ERROR_CRC;
}}
```

In **ecatslv.c**, lines 2110 to 2114, added:

```
DISABLE_ESC_INT();
```

ETHERCAT

In **ecatslv.c**, lines 2186 to 2190, added:

```
ENABLE_ESC_INT();
```

In **eoeappl.c**, lines 145-150, replaced:

```
switch ( ((ETHERNET_FRAME *) pFrame)->FrameType )
```

with:

```
switch ( SWAPWORD(((ETHERNET_FRAME *) pFrame)->FrameType ))
```

In **eoeappl.c**, lines 182 to 189, added:

```
else
{
  if(pSendFrame != NULL)
  {
    FREEMEM(pSendFrame);
    pSendFrame = NULL;
  }
}
```

In **eoeappl.c**, lines 298 to 305, replaced:

```
if ( SWAPWORD(pEoeInit->Flags1) & EOEINIT_CONTAINSMACADDR )
```

with:

```
pEoeInit->Flags1 = SWAPWORD(pEoeInit->Flags1);
               if (pEoeInit->Flags1 & EOEINIT_CONTAINSMACADDR )
```

In **esc.h**, lines 91 to 95, replaced:

```
#define ESC_EEPROM_ERROR_MASK                  0x7800
```

with:

```
#define ESC_EEPROM_ERROR_MASK                  0x6000
```

In **mailbox.h**, lines 101 to 114, replaced:

```
#define    MBX_OFFS_TYPE        0 /**< brief Protocol type offset*/
#define    MBX_MASK_TYPE        0x0F00 /**< brief Protocol type mask*/
#define    MBX_SHIFT_TYPE       8 /**< brief Protocol type shift*/
#define    MBX_OFFS_COUNTER     0 /**< brief Protocol counter offset*/
```

ETHERCAT

```
#define    MBX_MASK_COUNTER     0xF000 /**< brief Protocol counter mask*/
#define    MBX_SHIFT_COUNTER    12 /**< brief Protocol counter shift*/
```

with:

```
// ORIGINAL ^^^
#define    MBX_OFFS_TYPE        0 /**< brief Protocol type offset*/
#define    MBX_MASK_TYPE        0x000F /**< brief Protocol type mask*/
#define    MBX_SHIFT_TYPE       0 /**< brief Protocol type shift*/
#define    MBX_OFFS_COUNTER     0 /**< brief Protocol counter offset*/
#define    MBX_MASK_COUNTER     0x00F0 /**< brief Protocol counter mask*/
#define    MBX_SHIFT_COUNTER    4 /**< brief Protocol counter shift*/
```

In **objdef.c**, lines 890 to 895, replaced:

```
if((((UINT16)pVarPtr) & 0x1) == 0x1)
```

with:

```
if((((UINT32)pVarPtr) & 0x1) == 0x1)
```

In **sdoserv.c**, lines 508 to 513, replaced:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET] &= 0xFF00;
```

with:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET] &= 0x00FF;
```

In **sdoserv.c**, lines 527 to 540, replaced:

```
  pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
  |=     SDOHEADER_SIZEINDICATOR     | SDOHEADER_TRANSFERTYPE
  | completeAccess | ((MAX_EXPEDITED_DATA -
((UINT8)objLength)) << SDOHEADERSHIFT_DATASETSIZE) | SDOSERVICE_INITIATEUPLOADRES;
```

with:

```
  pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET] |= SWAPWORD(SDOHEADER_SIZEINDICATOR | SDOHEAD▶
ER_TRANSFERTYPE |completeAccess  | ((MAX_EXPEDITED_DATA - ((UINT8)objLength)) << SDOHEADERSHIFT_DATA▶
SETSIZE) | SDOSERVICE_INITIATEUPLOADRES);
```

ETHERCAT

In **sdoserv.c**, lines 555 to 564, replaced:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]    |=     SDOHEADER_SIZEINDICATOR     | completeAc▶
cess | SDOSERVICE_INITIATEUPLOADRES;
```

with:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
|= SWAPWORD(SDOHEADER_SIZEINDICATOR | completeAccess  | SDOSERVICE_INITIATEUPLOADRES);
```

In **sdoserv.c**, lines 572 to 577, replaced:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
|= SDOSERVICE_DOWNLOADSEGMENTRES;
```

with:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
|= SWAPWORD(SDOSERVICE_DOWNLOADSEGMENTRES);
```

In **sdoserv.c**, lines 583 to 588, replaced:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
|= SDOSERVICE_INITIATEDOWNLOADRES;
```

with:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
|= SWAPWORD(SDOSERVICE_INITIATEDOWNLOADRES);
```

In **sdoserv.c**, lines 597 to 602, replaced:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
= SDOSERVICE_ABORTTRANSFER;
```

with:

```
pSdoRes->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
= SWAPWORD(SDOSERVICE_ABORTTRANSFER);
```

In **sdoserv.c**, lines 629 to 634, replaced:

```
UINT8 sdoHeader = pSdoInd->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
  & SDOHEADER_COMMANDMASK;
```

ETHERCAT

with:

```
UINT8 sdoHeader = (pSdoInd->SdoHeader.Sdo[SDOHEADER_COMMANDOFFSET]
  & SDOHEADER_COMMANDMASK) >> SDOHEADER_COMMANDSHIFT;
```

In **sdoserv.c**, lines 658 to 672, replaced:

```
index = pSdoInd->SdoHeader.Sdo[SDOHEADER_INDEXHIOFFSET]
  & SDOHEADER_INDEXHIMASK;
      index <<= 8;
      index += pSdoInd->SdoHeader.Sdo[SDOHEADER_INDEXLOOFFSET]
     >> SDOHEADER_INDEXLOSHIFT;
/*the variable subindex contains the requested subindex of the SDO service */
 subindex    = pSdoInd->SdoHeader.Sdo[SDOHEADER_SUBINDEXOFFSET]
     >> SDOHEADER_SUBINDEXSHIFT;
```

with:

```
index = pSdoInd->SdoHeader.Sdo[SDOHEADER_INDEXHIOFFSET] & SDOHEADER_INDEXHIMASK;

index += (pSdoInd->SdoHeader.Sdo[SDOHEADER_INDEXLOOFFSET]
>> SDOHEADER_INDEXLOSHIFT) & SDOHEADER_INDEXLOMASK;

/*the variable subindex contains the requested subindex of the SDO service */

subindex    = (UINT8)(pSdoInd->SdoHeader.Sdo[SDOHEADER_SUBINDEXOFFSET] >>
SDOHEADER_SUBINDEXSHIFT) & SDOHEADER_SUBINDEXMASK;
```

In **sdoserv.h**, lines 94 to 116, replaced:

```
#define     SDOHEADER_COMMANDOFFSET            0 /**< brief Memory offset for the command*/
#define     SDOHEADER_INDEXLOOFFSET            0 /**< brief Memory offset for the low Byte of the ob▶
ject index*/
#define     SDOHEADER_INDEXHIOFFSET            1 /**< brief Memory offset for the high Byte of the ob▶
ject index*/
#define     SDOHEADER_SUBINDEXOFFSET           1 /**< brief Memory offset for subindex*/
#define     SDOHEADER_COMMANDMASK              0xFF /**< brief Mask to get the command Byte*/
#define     SDOHEADER_INDEXLOSHIFT             8 /**< brief Shift to get the low Byte of the ob▶
ject index*/
#define     SDOHEADER_INDEXHIMASK              0xFF /**< brief Mask to get the high byte of the ob▶
ject index*/
#define     SDOHEADER_SUBINDEXSHIFT            8 /**< brief Shift to get the subindex*/
```

with:

```
#define     SDOHEADER_COMMANDOFFSET            0
/**< brief Memory offset for the command*/
#define     SDOHEADER_INDEXLOOFFSET            0
/**< brief Memory offset for the low Byte of the object index*/
#define     SDOHEADER_INDEXHIOFFSET            1
/**< brief Memory offset for the high Byte of the object index*/
```

```
#define     SDOHEADER_SUBINDEXOFFSET            1
/**< brief Memory offset for subindex*/
#define     SDOHEADER_COMMANDMASK               0xFF00
/**< brief Mask to get the command Byte*/
#define     SDOHEADER_COMMANDSHIFT              8
/**< brief Shift to get the command Byte*/
#define     SDOHEADER_INDEXLOSHIFT              0
/**< brief Shift to get the low Byte of the object index*/
#define     SDOHEADER_INDEXLOMASK               0x00FF
/**< brief Mask to get low Byte of the object index*/
#define     SDOHEADER_INDEXHIMASK               0xFF00
/**< brief Mask to get the high byte of the object index*/
#define     SDOHEADER_SUBINDEXSHIFT             0
/**< brief Shift to get the subindex*/
#define     SDOHEADER_SUBINDEXMASK              0x00FF
/**< brief Mask to get the subindex*/
```

In coeappl.c, line 411, replaced

```
TOBJ1C00 sSyncmanagertype = {0x04, {0x0102, 0x0304}};
```

with:

```
TOBJ1C00 sSyncmanagertype = {0x04, {0x0201, 0x0403}};
```

Interrupt Enable and Disable

Depending on the SSC tool settings used, the user application may be required to implement the ENABLE_ESC_INT() and DISABLE_ESC_INT() functions. It is important that implementation of these functions does not completely disable all interrupts coming from the REM switch. The REM switch occasionally requests interrupts that are internally handled by the driver. The purpose of ENABLE_ESC_INT() and DISABLE_ESC_INT() is to enable and disable the REMS_EcatInt_AL_Event_Change, REMS_EcatInt_SYNC0_Event, and REMS_EcatInt_SYNC1_Event events. As long as these events are disabled after a call to DISABLE_ESC_INT() and enabled after a call to ENABLE_ESC_INT(), the user system behaves properly.

Application Programming Interface (API) Usage

In the application, a maximum of two operational threads are allowed. One thread can act as the standard, background thread. This thread can use all API functions except those ending in Isr. The other thread is limited to API functions ending in Isr. The EtherCAT follower stack meets this requirement. If using the EtherCAT follower stack, it is recommended to call MainLoop() in the background of the application and call PDI_Isr(), Sync0_Isr(), and Sync1_Isr() from the interrupt handler. PDI_Isr(), Sync0_Isr(), and Sync1_Isr() only use API functions that end in Isr.

POWERLINK

To set up a POWERLINK device, a TCP/IP protocol stack and a POWERLINK follower stack are necessary. It is recommended to use the open-source openPOWERLINK stack. This driver package is developed and tested using this POWERLINK follower stack.

This REM switch driver and firmware software is developed with the REM switch hardware and the openPOWERLINK stack to handle all aspects of the POWERLINK protocol while operating as a controlled node (CN). The operating features include operation as a 100 Mbps, half-duplex, two-port hub and REM switch firmware and hardware assisted poll request (PREQ) and poll response (PRES) autoresponse.

In addition, it is possible to add standard TCP/IP features, such as a web server. Because the POWERLINK protocol tightly controls when traffic is allowed to enter the network, the openPOWERLINK stack primarily manages this operation with the REM switch firmware providing autoresponse features during the asynchronous phase.

POWERLINK INITIALIZATION

To initialize a POWERLINK device, perform the following actions:

1. Configure the external interrupt input pins on the host processor.
2. Assert the REM switch reset line and wait for it to become ready.
3. Call REMS_StdInit() and provide the PHY mode (currently MII), the clock enable flag (set to 0), and the REM switch interrupt lines for use as the three priority levels.
4. Call REMS_StdAssignInterrupt and REMS_StdEnableInterrupt to establish the REM switch hardware interrupt configuration:

```
REMS_StdAssignInterrupt(REMS_Int_Port_1_Link_Change, REMS_Int_Line_0);
REMS_StdAssignInterrupt(REMS_Int_Port_2_Link_Change, REMS_Int_Line_0);
REMS_StdAssignInterrupt(REMS_Int_Queue_0_Packet_Ready, REMS_Int_Line_1);
REMS_StdEnableInterrupt(REMS_Int_Port_1_Link_Change);
REMS_StdEnableInterrupt(REMS_Int_Port_2_Link_Change);
REMS_StdEnableInterrupt(REMS_Int_Queue_0_Packet_Ready);
```

5. Call REMS_StdSetMacAddress() to set the MAC address for the system. There are three MAC address parameters for this function. For POWERLINK, supply the same MAC address for each of the MAC address parameters.
6. Call REMS_StdSetPortState() for each of the switch ports with REMS_PORT_FORWARDING and the port number as the arguments.

POWERLINK INTERRUPT HANDLING

When REMS_StdInit() is called, the interrupt outputs are assigned high, medium, and low priority positions according to the passed in parameters. The REM switch driver assigns interrupt sources to interrupt lines as shown in Table 10.

Table 10. Interrupt Priorities for POWERLINK

Priority	Event	Purpose
Low	REMS_StdInt_Port_1_LinkChange	Port 1 link up or down
Low	REMS_StdInt_Port_2_LinkChange	Port 2 link up or down
High	REMS_Int_Queue_0_Packet_Ready	Packet received

The low-level interrupt handler must be aware of which REM switch interrupt line caused the interrupt. The low-level interrupt handler may then call the REM switch event handler function. After this function is called, all pending interrupts are evaluated and can be handled individually in a loop. The recommended REM switch interrupt handler for POWERLINK is as follows:

```
/* Low level hardware ISR's: */
void REMS_Interrupt_Line0(void)
{
  HandleIntREMS(0);
}

void REMS_Interrupt_Line1(void)
{
  HandleIntREMS(1);
}
```

```
void REMS_Interrupt_Line2(void)
{
  HandleIntREMS(2);
}

/* Common code to handle interrupts: */
void HandleIntREMS(REMS_IntLine_t line)
{
  tOplkError rv;
  REMS_CommonEnetPort_t   rcvPort;
  static unsigned char dummyBuffer[1500];
  static int dummyCount;

  REMS_stdIntEvent_t event;

  REMS_StdEvaluateInterrupt(line);
  do {
    event = REMS_StdGetNextEvent(line);
    switch (event) {
      case REMS_StdInt_Port_1_LinkChange:
      case REMS_StdInt_Port_2_LinkChange:
        /* .... manage PHY link state as described below .... */
        break;

      case REMS_Int_Queue_0_Packet_Ready:
          rv = edrv_receiveBuffer();
          if (rv) {
            // ERROR CASE, read the frame and dump it...
            REMS_StdReadPacket(&rcvPort, &dummyBuffer, &dummyCount);
          }
          break;

      /* Add other interrupt event cases here as needed */

      default:
        break;
    }
  } while (event != REMS_Int_None);
}
loop.
```

POWERLINK PHY LINK STATE INTERRUPT HANDLING

When the REM switch detects a link up or down event from the PHY (because the link signal output from the PHY is required to be connected to REM switch link status input), the link speed and duplex settings must be written to the REM switch by the host processor. The REM switch does not determine these settings on its own. As a result, when the Ethernet link is functional (when either the REMS_StdInt_Port_1_LinkChange or the REMS_StdInt_Port_2_LinkChange interrupt event occurs), read the link speed, duplex from the PHYs, and write to the REM switch using REMS_StdSetSpeedAndDuplex(). In the case of POWERLINK, fix the speed and duplex at 100 Mbps half duplex operation. An error is likely to occur if the PHY reports something other than 100 Mbps half duplex.

POWERLINK RECEIVED PACKET INTERRUPT HANDLING

When a packet is received, the REM switch issues a REMS_Int_Queue_0_Packet_Ready interrupt to the processor. In response, call edrv_receiveBuffer(). This function asks the POWERLINK stack for a buffer and, using REMS_StdReadPacket(), reads the packet from the

hardware. The remainder of the function is responsible for passing the packet into the stack logic, at which point stack logic takes over control of functionality.

Functions such as edrv_receiveBuffer() are not typically present in the standard REMs driver package, because it is specific to openPOWERLINK and is one part of a porting layer the user must provide. An example version is provided in the **porting_layer_helpers** directory for **edrv-fido1100.c**. There are several functions contained in this file that the user may use.

POWERLINK PACKET TRANSMISSION

Because the REM switch POWERLINK implements an autoresponse, POWERLINK is always monitoring received packets. If POWERLINK detects a frame that requires an immediate response, it provides a response with no software intervention. This response allows for a very fast response but requires that the response frame must already be on-board. The openPOWERLINK stack accounts for this factor and uses the edrv_sendTxBuffer() and edrv_updateTxBuffer() functions. Transmission of frames to the REM switch device is still accomplished with REMS_StdXmitPacket().

POWERLINK Dynamic Node Allocation

Many POWERLINK controllers that exist in the industry allow a user to dynamically assign node numbers, such as the following:

- Int REMS_Powerlink_SetPortEnableMask(uint8_t mask)

  Description: according to the EPSG Draft Standard 302-E, the DNA-supported device must have the ability to enable or disable packet forwarding. REMS_Powerlink_SetPortEnableMask() can be used for this purpose.

  Argument: what to put in as an argument is an 8-bit number with the last two bits being set as follows in Table 11.

  Return type: REMS_OK for success.

  Table 11. Mask

Bit 1	Bit 0	Packet Forwarding Result
0	0	Disabled
0	1	Enabled from Port 0
1	0	Enabled from Port 1
1	1	Enabled for both ports

- Int REMS_Powerlink_SetPResChainingMode(uint8_t presMode)

  Description: as per EPSG Draft Standard 302-C specifications, a poll response chaining-supported device must enable/disable poll response chaining based on presMode in the sync request frame. To achieve this, the REMS_Powerlink_SetPResChainingMode() API can be used.

  Argument: if the value of presMode is 0x00, poll response chaining mode is disabled. If it is 0x01, the mode is enabled.

  Return type: REMS_OK for success.

  Table 12. Mask

Bit 1	Bit 0	Packet Forwarding Result
0	0	Disabled
0	1	Enabled

- Int REMS_Powerlink_SetPRes_Delay(uint32_t delayFirst_ns, uint32_t delaySecond_ns)

  Description: according to the EPSG Draft Standard 302-C specifications, the sync request frame includes PResTimeFirst and PResTimeSecond values. These values are important for the device to determine the timing to send the PRes frame in poll response chaining. To configure these values on the POWERLINK device, the REMS_Powerlink_SetPRes_Delay() API can be used.

  Argument:

  - delayFirst_ns: contains the value of PRes Response Time [ns] starting from the end of the PResMN
  - delaySecond_ns: contains the value of PRes Response Time [ns] for the secondary direction of communication starting from the end of the PResMN. This is valid for the device that supports ring redundancy along with poll response chaining.

POWERLINK

Return type: REMS_OK for success.

- Int REMS_Powerlink_SetCrossChannelNodeID(uint32_t channel, uint8_t id)

  Description: if the POWERLINK device that supports poll response chaining has an RxPDO, then that device must be configured with one of the cross traffic channels with ID 240. To achieve this, the REMS_Powerlink_SetCrossChannelNodeID() API can be used. If the device does not have any RxPDO, then these configurations changes must be ignored.

  Argument:

  - Channel: there are three available channels for cross channel traffic, for example, 0, 1, or 2.
  - Id: for poll response chaining, the ID must be set to 240. If it is set to 0, then the corresponding channel is disabled.

  Return type: REMS_OK for success.

REGISTER MAPS AND DEFINITIONS

Table 13 and Table 14 detail the register mapping and definitions for the fido5100 and fido5200. The user of the REM switch typically does not need to modify any values in these registers because the API described in this user guide interacts with the registers as required. Note that the register maps and definitions provided in this user guide are provided as reference. It is not intended for the user to read and write registers in the exact ways defined, but to use the API defined in the driver source code.

For direct address registers, when accessing registers, the contents are swapped when read and unswapped when written. When accessing memory, the contents are swapped when read or written. In addition, for EtherCAT, there are many other registers that are not listed in Table 13 and Table 14. For these registers, you may consult the ET1100 data sheet from Beckhoff.

Table 13. Direct Address Register Definitions

Region	Address	Name	Description	Reset	Register Width	Access
Host Registers	0x00	Host Queue 0 read register	Host priority read queues	0x00000000	16/32[1]	R
Host Registers	0x00	Host Queue 0 write register	Host priority write queues	N/A[2]	16/32[1]	W
Host Registers	0x04	Host Queue 1 read register	Host priority read queues	0x00000000	16/32[1]	R
Host Registers	0x04	Host Queue 1 write register	Host priority write queues	N/A[2]	16/32[1]	W
Host Registers	0x08	Host Queue 2 read register	Host priority read queues	0x00000000	16/32[1]	R
Host Registers	0x08	Host Queue 2 write register	Host priority write queues	N/A[2]	16/32[1]	W
Host Registers	0x0C	Host Queue 3 read register	Host priority read queues	0x00000000	16/32[1]	R
Host Registers	0x0C	Host Queue 3 write register	Host priority write queues	N/A[2]	16/32[1]	W
Reserved	0x10 to 0x14	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x18	Host Read Queue 0 data head	Host controlled queues	0x00000000	16/32[1]	R
Host Registers	0x18	Host Write Queue 0 data head	Host controlled queues	N/A[2]	16/32[1]	W
Host Registers	0x1C	Host Read Queue 1 data head	Host controlled queues	0x00000000	16/32[1]	R
Host Registers	0x1C	Host Write Queue 1 data head	Host controlled queues	N/A[2]	16/32[1]	W
Host Registers	0x20	Queue status register	Interrupt management	0x7F00	16	R
Host Registers	0x24	Timer status register	Interrupt management	0x0000	16	R/W
Host Registers	0x28	Universal input/output controller (UIC) interrupt status register	Interrupt management	0x0000	16	R/W
Host Registers	0x2C	Composite interrupt status register	Interrupt management	0x0000	16	R/W
Host Registers	0x30	Host indirect address register	Register map	0x00	16	R/W
Host Registers	0x34	Host indirect read data register	Register map	N/A[2]	16	R
Host Registers	0x34	Host indirect write data register	Register map	N/A[2]	16	W
Host Registers	0x38	Host Write Queue 1 completion register	Host controlled queues	0x00	16	R
Host Registers	0x3C	Host Write Queue 0 completion register	Host controlled queues	0x00	16	R

[1] Registers with a variable width are memory queues and their width is dependent on the host interface of the fido5100 or fido5200.

[2] N/A means not applicable.

Table 14. Indirect Address Host Register Definitions

Region	Address	Name	Definition	Reset Value	Register Width	Access
Host Registers	0x00	Host control register	Host control register	0x6200	16	R/W
Host Registers	0x01	Protocol register	Switch type register	NVM[1]	16	R
Host Registers	0x02	Version register	ID register	0x0531	16	R
Host Registers	0x03	Part number register	ID register	0x3300	16	R
Host Registers	0x04	Original equipment manufacturer (OEM) ID register	Switch type register	NVM[1]	16	R
Host Registers	0x05	Host space reservation register	Host controlled queues	0x0000	8	R/W
Host Registers	0x06	Buffer list stack pointer	Buffer management	0x0000	8	R
Host Registers	0x07	Buffer list max stack index	Buffer management	0x0000	8	R
Host Registers	0x08	Port configuration register	Port interface	N/A[2]	16	R/W
Host Registers	0x09	Port cyclical redundancy check (CRC) offset register	Port interface	0x0000	16	W
Host Registers	0x0A	Port 1 CRC16 header register	Port interface	0x0000	16	R/W
Host Registers	0x0B	Port 2 CRC16 header register	Port interface	0x0000	16	R

REGISTER MAPS AND DEFINITIONS

Table 14. Indirect Address Host Register Definitions (Continued)

Region	Address	Name	Definition	Reset Value	Register Width	Access
Host Registers	0x0C	Port 1 CRC16 start register	Port interface	0x0000	16	R/W
Host Registers	0x0D	Port 2 CRC16 start register	Port interface	0x0000	16	R/W
Host Registers	0x0E	Activity LED control register	LED requirements	0x0000	16	R/W
Reserved	0x0F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Reserved	0x10 to 0x17	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x18	Host Write Queue 0 address register	Host controlled queues	0x0000	16	R/W
Host Registers	0x19	Host Read Queue 0 address register	Host controlled queues	0x0000	16	R/W
Host Registers	0x1A	Host Write Queue 1 address register	Host controlled queues	0x0000	16	R/W
Host Registers	0x1B	Host Read Queue 1 address register	Host controlled queues	0x0000	16	R/W
Host Registers	0x1C	Host read queue control	Host priority read queues	0x0000	16	R/W
Reserved	0x1D	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x1E	Host write queue control	Host priority write queues	0x0000	16	R/W
Reserved	0x1F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x20	Atomic Communication Register 0	Atomic communication registers	0x0000	16	R/W
Host Registers	0x21	Atomic Communication Register 1	Atomic communication registers	0x0000	16	R/W
Host Registers	0x22	Atomic Communication Register 2	Atomic communication registers	0x0000	16	R/W
Host Registers	0x23	Atomic Communication Register 3	Atomic communication registers	0x0000	16	R/W
Host Registers	0x24	Atomic Communication Register 4	Atomic communication registers	0x0000	16	R/W
Host Registers	0x25	Atomic Communication Register 5	Atomic communication registers	0x0000	16	R/W
Host Registers	0x26	Atomic Communication Register 6	Atomic communication registers	0x0000	16	R/W
Host Registers	0x27	Atomic Communication Register 7	Atomic communication registers	0x0000	16	R/W
Host Registers	0x28	Atomic writer status register	Atomic communication registers	0x0000	16	R/W
Reserved	0x29 to 0x2F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x30	Queue interrupt mask register	Interrupt management	0x0000	16	R/W
Reserved	0x31	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x32	Timer interrupt setup register	Interrupt management	0x0000	16	R/W
Host Registers	0x33	Timer control unit interrupt setup register	Interrupt management	0x0000	16	R/W
Host Registers	0x34	Timer interrupt mask register	Interrupt management	0x0000	16	R/W
Host Registers	0x35	Link status interrupt setup register	Interrupt management	0x0000	16	R/W
Host Registers	0x36	UIC interrupt setup register	Interrupt management	0x0000	16	R/W
Host Registers	0x37	UIC interrupt mask register	Interrupt management	0x0000	16	R/W
Reserved	0x38 to 0x39	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Host Registers	0x3A	Host queue packet ready interrupt setup register	Interrupt management	0x0000	16	R/W
Host Registers	0x3B	Host queue space available interrupt setup register	Interrupt management	0x0000	16	R/W
Reserved	0x3C to 0x3F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Timers	0x40	Time control unit control register	TCU control register	0x0000	16	R/W
Timers	0x41	Timer Register 0, Bits[15:0]	Timer register	0x0000	16	R/W, Auto-Increment (AINC) 0
Timers	0x41	Timer Register 1, Bits[31:16]	Timer register	0x0000	16	R/W, AINC 1
Timers	0x41	Timer Register 2, Bits[47:32]	Timer register	0x0000	16	R/W, AINC 2
Timers	0x41	Timer Register 3, Bits[63:48]	Timer register	0x0000	16	R/W, AINC 3
Timers	0x42	Timer Addend Register A 0, Bits[15:0]	Timer Addend Register A	0x0000	16	R/W, AINC 0
Timers	0x42	Timer Addend Register A 1, Bits[31:16]	Timer Addend Register A	0x8000	16	R/W, AINC 1

REGISTER MAPS AND DEFINITIONS

Table 14. Indirect Address Host Register Definitions (Continued)

Region	Address	Name	Definition	Reset Value	Register Width	Access
Timers	0x43	Timer Addend Register B count, Bits[15:0]	Timer Addend Register B count	0x0000	16	R/W, AINC 0
Timers	0x43	Timer Addend Register B count, Bits[31:16]	Timer Addend Register B count	0x0000	16	R/W, AINC 1
Timers	0x44	Timer Addend Register B 0, Bits[15:0]	Timer Addend Register B	0x0000	16	R/W, AINC 0
Timers	0x44	Timer Addend Register B 1, Bits[31:16]	Timer Addend Register B	0x0000	16	R/W, AINC 1
Timers	0x45	I/O control register	Timer I/O control register	0x0000	16	R/W
Timers	0x46	TCU start compare register	TCU start register	0xFFFFFFFF	32	W, AINC
Timers	0x47	Port 1 delay register	P1 delay	0x0000	16	R/W
Timers	0x48	Port 2 delay register	P2 delay	0x0000	16	R/W
Timers	0x49	Timer Periodic Interrupt 0	Timer Periodic Interrupt 0	0x00000000	32	W, AINC
Timers	0x4A	Timer Periodic Interrupt 1	Timer Periodic Interrupt 1	0x00000000	32	W, AINC
Timers	0x4B	Port 1 Egress Time Register B 0, Bits[0:15]	Egress time register	0x0000	16	R, AINC 0
Timers	0x4B	Port 1 Egress Time Register B 1, Bits[16:31]	Egress time register	0x0000	16	R, AINC 1
Timers	0x4B	Port 1 Egress Time Register B 2, Bits[32:47]	Egress time register	0x0000	16	R, AINC 2
Timers	0x4B	Port 1 Egress Time Register B 3, Bits[48:63]	Egress time register	0x0000	16	R, AINC 3
Timers	0x4C	Port 2 Egress Time Register B 0, Bits[0:15]	Egress time register	0x0000	16	R, AINC 0
Timers	0x4C	Port 2 Egress Time Register B 1, Bits[16:31]	Egress time register	0x0000	16	R, AINC 1
Timers	0x4C	Port 2 Egress Time Register B 2, Bits[32:47]	Egress time register	0x0000	16	R, AINC 2
Timers	0x4C	Port 2 Egress Time Register B 3, Bits[48:63]	Egress time register	0x0000	16	R, AINC 3
Timers	0x4D	Time Control Unit Memory Register A	Timer control unit memory register	N/A[2]	16	W
Timers	0x4E	Time Control Unit Memory Register B	Timer control unit memory register	N/A[2]	16	W
Timers	0x4F	Timer control unit error addend register	Timer control unit error addend	0x0000	16	R/W
Timers	0x50	Timer Input Capture Register A 0, Bits[15:0]	Timer input capture register	0x0000	16	R, AINC 0
Timers	0x50	Timer Input Capture Register A 1, Bits[31:16]	Timer input capture register	0x0000	16	R, AINC 1
Timers	0x50	Timer Input Capture Register A 2, Bits[47:32]	Timer input capture register	0x0000	16	R, AINC 2
Timers	0x50	Timer Input Capture Register A 3, Bits[63:48]	Timer input capture register	0x0000	16	R, AINC 3
Timers	0x51	Timer Input Capture Register B 0, Bits[15:0]	Timer input capture register	0x0000	16	R, AINC 0
Timers	0x51	Timer Input Capture Register B 1, Bits[31:16]	Timer input capture register	0x0000	16	R, AINC 1
Timers	0x51	Timer Input Capture Register B 2, Bits[47:32]	Timer input capture register	0x0000	16	R, AINC 2
Timers	0x51	Timer Input Capture Register B 3, Bits[63:48]	Timer input capture register	0x0000	16	R, AINC 3
Timers	0x52	Timer Input Capture Register C 0, Bits[15:0]	Timer input capture register	0x0000	16	R, AINC 0
Timers	0x52	Timer Input Capture Register C 1, Bits[31:16]	Timer input capture register	0x0000	16	R, AINC 1
Timers	0x52	Timer Input Capture Register C 2, Bits[47:32]	Timer input capture register	0x0000	16	R, AINC 2
Timers	0x52	Timer Input Capture Register C, Bits[63:48]	Timer input capture register	0x0000	16	R, AINC 3

REGISTER MAPS AND DEFINITIONS

Table 14. Indirect Address Host Register Definitions (Continued)

Region	Address	Name	Definition	Reset Value	Register Width	Access
Timers	0x53	Timer Input Capture Register D 0, Bits[15:0]	Timer input capture register	0x0000	16	R, AINC 0
Timers	0x53	Timer Input Capture Register D 1, Bits[31:16]	Timer input capture register	0x0000	16	R, AINC 1
Timers	0x53	Timer Input Capture Register D 2, Bits[47:32]	Timer input capture register	0x0000	16	R, AINC 2
Timers	0x53	Timer Input Capture Register D 3, Bits[63:48]	Timer input capture register	0x0000	16	R, AINC 3
Timers	0x54	Timer Output Compare Register A 0, Bits[15:0]	Timer output compare register	N/A[2]	16	W, AINC 0
Timers	0x54	Timer Output Compare Register A 1, Bits[31:16]	Timer output compare register	N/A[2]	16	W, AINC 1
Timers	0x54	Timer Output Compare Register A 2, Bits[47:32]	Timer output compare register	N/A[2]	16	W, AINC 2
Timers	0x54	Timer Output Compare Register A 3, Bits[63:48]	Timer output compare register	N/A[2]	16	W, AINC 3
Timers	0x55	Timer Output Compare Register B 1, Bits[15:0]	Timer output compare register	N/A[2]	16	W, AINC 0
Timers	0x55	Timer Output Compare Register B 1, Bits[31:16]	Timer output compare register	N/A[2]	16	W, AINC 1
Timers	0x55	Timer Output Compare Register B 2, Bits[47:32]	Timer output compare register	N/A[2]	16	W, AINC 2
Timers	0x55	Timer Output Compare Register B 3, Bits[63:48]	Timer output compare register	N/A[2]	16	W, AINC 3
Timers	0x56	Timer Output Compare Register C 0, Bits[15:0]	Timer output compare register	N/A[2]	16	W, AINC 0
Timers	0x56	Timer Output Compare Register C 1, Bits[31:16]	Timer output compare register	N/A[2]	16	W, AINC 1
Timers	0x56	Timer Output Compare Register C 2, Bits[47:32]	Timer output compare register	N/A[2]	16	W, AINC 2
Timers	0x56	Timer Output Compare Register C 3, Bits[63:48]	Timer output compare register	N/A[2]	16	W, AINC 3
Timers	0x57	Timer Output Compare Register D 0, Bits[15:0]	Timer output compare register	N/A[2]	16	W, AINC 0
Timers	0x57	Timer Output Compare Register D 1, Bits[31:16]	Timer output compare register	N/A[2]	16	W, AINC 1
Timers	0x57	Timer Output Compare Register D 2, Bits[47:32]	Timer output compare register	N/A[2]	16	W, AINC 2
Timers	0x57	Timer Output Compare Register D 3, Bits[63:48]	Timer output compare register	N/A[2]	16	W, AINC 3
Timers	0x58	Port 1 Egress Time Register A 0, Bits[0:15]	Egress time register	0x0000	16	R, AINC 0
Timers	0x58	Port 1 Egress Time Register A 1, Bits[16:31]	Egress time register	0x0000	16	R, AINC 1
Timers	0x58	Port 1 Egress Time Register A 2, Bits[32:47]	Egress time register	0x0000	16	R, AINC 2
Timers	0x58	Port 1 Egress Time Register A 3, Bits[48:63]	Egress time register	0x0000	16	R, AINC 3
Timers	0x59	Port 2 Egress Time Register A 0, Bits[0:15]	Egress time register	0x0000	16	R, AINC 0
Timers	0x59	Port 2 Egress Time Register A 1, Bits[16:31]	Egress time register	0x0000	16	R, AINC 1
Timers	0x59	Port 2 Egress Time Register A 2, Bits[32:47]	Egress time register	0x0000	16	R, AINC 2

REGISTER MAPS AND DEFINITIONS

Table 14. Indirect Address Host Register Definitions (Continued)

Region	Address	Name	Definition	Reset Value	Register Width	Access
Timers	0x59	Port 2 Egress Time Register A 3, Bits[48:63]	Egress time register	0x0000	16	R, AINC 3
Timers	0x5A	Port 1 peer delay register [15:0]	Port interface	0x0000	16	W, AINC 0
Timers	0x5A	Port 1 peer delay register [31:16]	Port interface	0x0000	16	W, AINC 1
Timers	0x5A	Port 1 peer delay register [47:32]	Port interface	0x0000	16	W, AINC 2
Timers	0x5A	Port 1 peer delay register [63:48]	Port interface	0x0000	16	W, AINC 3
Timers	0x5B	Port 2 peer delay register [15:0]	Port interface	0x0000	16	W, AINC 0
Timers	0x5B	Port 2 peer delay register [31:16]	Port interface	0x0000	16	W, AINC 1
Timers	0x5B	Port 2 peer delay register [47:32]	Port interface	0x0000	16	W, AINC 2
Timers	0x5B	Port 2 peer delay register [63:48]	Port interface	0x0000	16	W, AINC 3
Timers	0x5C	Timer update compare register, Bits[15:0]	Timer update compare register	0xCA00	16	R/W, AINC 0
Timers	0x5C	Timer update compare register, Bits[31:16]	Timer update compare register	0x3B9A	16	R/W, AINC 1
Reserved	0x5D to 0x5F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
EtherCAT Lookup	0x60	Address lookup control register	EtherCAT lookup interface	0x0000	16	R/W
EtherCAT Lookup	0x61	Max valid address register	EtherCAT lookup interface	0x3000	16	R/W
EtherCAT Lookup	0x62	Write offset register	EtherCAT lookup interface	0x0000	16	R/W
EtherCAT Lookup	0x63	FMMU list programming register	EtherCAT lookup interface	N/A[2]	16	W
EtherCAT Lookup	0x64	Sync manager programming register	EtherCAT lookup interface	N/A[2]	16	W
EtherCAT Lookup	0x65	Sync manager control/state register	EtherCAT lookup interface	N/A[2]	16	W
EtherCAT Lookup	0x66	Sync manager program index register	EtherCAT lookup interface	N/A[2]	16	W
EtherCAT Lookup	0x67	Sync manager atomic status register	EtherCAT lookup interface	N/A[2]	16	R
EtherCAT Lookup	0x68	Sync Manager 0 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x69	Sync Manager 1 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6A	Sync Manager 2 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6B	Sync Manager 3 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6C	Sync Manager 4 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6D	Sync Manager 5 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6E	Sync Manager 6 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
EtherCAT Lookup	0x6F	Sync Manager 7 atomic read/write register	EtherCAT lookup interface	0x00	16	R/W
Reserved	0x70 to 0x7F	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 1	0x80	Interframe gap set register	MAC interframe gap set register	0x0012	16	R/W
MAC 1	0x81	Full duplex control register	MAC full duplex register	0x0001	16	R/W
MAC 1	0x82	Maximum retry register	MAC maximum retry register	0x0010	16	R/W
Reserved	0x83	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]

REGISTER MAPS AND DEFINITIONS

Table 14. Indirect Address Host Register Definitions (Continued)

Region	Address	Name	Definition	Reset Value	Register Width	Access
MAC 1	0x84	CRC check enable register	MAC CRC check enable register	0x0001	16	R/W
MAC 1	0x85	Receive interframe gap control register	MAC receive interframe gap set register	0x0012	16	R/W
MAC 1	0x86	Maximum receive length register	MAC receive maximum length register	0x2710	16	R/W
MAC 1	0x87	Minimum receive length register	MAC receive minimum length register	0x0040	16	R/W
Reserved	0x88	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 1	0x89	Statistic register read register (LSB)	MAC CPU read data out (DOUT) low register, Bits[15:0]	N/A[2]	16	R, AINC 0
MAC 1	0x89	Statistic register read register (MSB)	MAC CPU read DOUT high register, Bits[31:16]	N/A[2]	16	R, AINC 1
Reserved	0x8A	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Reserved	0x8B	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 1	0x8C	Speed register	MAC speed register	0x0004	16	R/W
Reserved	0x8D to 0xAF	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 2	0xB0	Interframe gap set register	MAC interframe gap set register	0x0012	16	R/W
MAC 2	0xB1	Full duplex control register	MAC full duplex register	0x0001	16	R/W
MAC 2	0xB2	Max retry register	MAC maximum retry register	0x0010	16	R/W
Reserved	0xB3	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 2	0xB4	CRC check enable register	MAC CRC check enable register	0x0001	16	R/W
MAC 2	0xB5	Receive interframe gap control register	MAC receive interframe gap set register	0x0012	16	R/W
MAC 2	0xB6	Maximum receive length register	MAC receive maximum length register	0x2710	16	R/W
MAC 2	0xB7	Minimum receive length register	MAC receive minimum length register	0x0040	16	R/W
Reserved	0xB8	N/A[2]	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 2	0xB9	Statistic register read register (LSB)	MAC CPU read DOUT low register, Bits[15:0]	Not applicable	16	R, AINC 0
MAC 2	0xB9	Statistic register read register (MSB)	MAC CPU read DOUT high register, Bits[31:16]	N/A[2]	16	R, AINC 1
Reserved	0xBA	Spare MAC register	N/A[2]	N/A[2]	N/A[2]	N/A[2]
Reserved	0xBB	Spare MAC register	N/A[2]	N/A[2]	N/A[2]	N/A[2]
MAC 2	0xBC	Speed register	MAC speed register	0x0004	16	R/W
Reserved	0xBD to 0xFF	Spare	N/A[2]	N/A[2]	N/A[2]	N/A[2]

[1] NVM means nonvolatile memory.

[2] N/A means not applicable.

DIRECT ADDRESS REGISTERS

Table 15. Host Queue n Read Registers—Address 0x00, Address 0x04, Address 0x08, and Address 0x0C

Bit 31 to Bit 0
Data

Reads from the direct address registers retrieve data from the queue that was read from the memory and pointed to by the current value of the read queue address register. These registers are read only. Writes have no effect.

Table 16. Host Queue n Write Registers—Address 0x00, Address 0x04, Address 0x08, and Address 0x0C

Bit 31 to Bit 0
Data

REGISTER MAPS AND DEFINITIONS

Table 17. Host Read Queue n Data Head—Address 0x18 and Address 0x1C

Byte A	Byte B	Byte C	Byte D
Bit 31 to Bit 24	Bit 23 to Bit 16	Bit 15 to Bit 8	Bit 7 to Bit 0
Queue Head Data			

Register 0x18 and Register 0x1C return the data currently at the head of the corresponding queue. There is special handling for this data based on bus width and Endian selection, shown in Table 18.

Table 18. Host Read Queue Data Head Special Handling

Size_32	Little Endian (LE)	Bus Description	Handling[1]
0	0	16-bit big endian	Return [A:B] on first access, [C:D] on second access
0	1	16-bit little endian	Return [B:A] on first access, [D:C] on second access
1	0	32-bit big endian	Return [A:B:C:D]
1	1	32-bit little endian	Return [D:C:B:A]

[1] A colon indicates the order in which the data is read. For instance, A:B means that Byte A is read before Byte B.

Table 19. Host Write Queue n Data Head—Address 0x18 and Address 0x1C

Byte A	Byte B	Byte C	Byte D
Bit 31 to Bit 24	Bit 23 to Bit 16	Bit 15 to Bit 8	Bit 7 to Bit 0
Queue Head Data			

The host writes Register 0x18 and Register 0x1C, and data is forwarded to switch memory as described in Table 18. There is special handling for this data based on bus width and Endian selection, as shown in Table 20.

Table 20. Host Write Queue Data Head Special Handling

Size_32	LE	Bus Description	Handling[1]
0	0	16-bit big endian	Write [A:B] on first access, write [C:D] on second access
0	1	16-bit little endian	Write [B:A] on first access, write [D:C] on second access
1	0	32-bit big endian	Write [A:B:C:D]
1	1	32-bit little endian	Write [D:C:B:A]

[1] A colon indicates the order in which the data is read. For instance, A:B means that Byte A is read before Byte B.

Table 21. Queue Status Register—Address 0x20

Bit 15 to Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7 to Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Q3 R	Q2 R	Q1 R	Q0 R	Reserved	Q3 NE	Q2 NE	Q1 NE	Q0NE

Table 22. Bit Descriptions for Queue Status Register

Bits	Bit Name	Settings	Description
[15:12]	Reserved		Reserved.
[11:8]	Qx R		Queue 0 to Queue 3 are ready.
		0	Packet data is not ready to be written.
		1	Packet data is ready to be written.
[7:4]	Reserved		Reserved.
[3:0]	Qx NE		Queue 0 to Queue 3 not empty. This bit indicates that there is data that has yet to be read from the corresponding queue.
		0	No packets or buffers ready to read from the queue.
		1	Packet in progress, or a packet is waiting to be read.

Table 23. Timer Status Register, Upper Byte—Address 0x24

Bit 15	Bit 14	Bit 13	Bit 12 to Bit 11	Bit 10	Bit 9	Bit 8
TCU Interrupt 2	TCU Interrupt 1	TCU Interrupt 0	TCU Pin 3 event	TCU Pin 2 event	TCU Pin 1 event	TCU Pin 0 event

REGISTER MAPS AND DEFINITIONS

Table 24. Timer Status Register, Lower Byte—Address 0x24

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Compare 3	Compare 2	Compare 1	Compare 0	Capture 3 event	Capture 2 event	Capture 1 event	Capture 0 event

Table 25. UIC Interrupt Status Register, Upper Byte—Address 0x28

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11 to Bit 8
Port 2 link status change	Port 1 link status change	Timer Periodic Interrupt 1	Timer Periodic Interrupt 0	Reserved

Table 26. UIC Interrupt Status Register, Lowe Byte—Address 0x28

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Host UIC Interrupt 3	Host UIC Interrupt 2	Host UIC Interrupt 1	Host UIC Interrupt 0	Port 2 UIC Interrupt 1	Port 2 UIC Interrupt 0	Port 1 UIC Interrupt 1	Port 1 UIC Interrupt 0

Table 27. Composite Interrupt Status Register, Upper Byte—Address 0x2C

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Reserved	Interrupt Line 2 disable	Interrupt Line 1 disable	Interrupt Line 0 disable	Reserved	Interrupt Line 2 asserted	Interrupt Line 1 asserted	Interrupt Line 0 asserted

Table 28. Composite Interrupt Status Register, Lower Byte—Address 0x2C

Bit 7 to Bit 3	Bit 2	Bit 1	Bit 0
Reserved	UIC interrupt active	Timer interrupt active	Queue interrupt active

Register 0x2C is R/W. When a write occurs, Bit 12 to Bit 14 are modified and they are only modified if the written bit is a 1.

Table 29. Bit Descriptions for Composite Interrupt Status Register

Bits	Bit Name	Settings	Description
15	Reserved		Reserved.
[14:12]	Interrupt Line x disable		Interrupt Line x disable. These bits are reset to 0 and toggled by writing 1.
		0	Interrupt line operates normally, matching the state of the corresponding interrupt Line x asserted.
		1	Interrupt line is deasserted, regardless of interrupt status.
11	Reserved		Reserved.
10	Interrupt Line 2 asserted		Interrupt Line 2 asserted. This bit mirrors the status of the line before the disable mask.
9	Interrupt Line 1 asserted		Interrupt Line 1 asserted. This bit mirrors the status of the line before the disable mask.
8	Interrupt Line 0 asserted		Interrupt Line 0 asserted. This bit mirrors the status of the interrupt before the disable mask.
[7:3]	Reserved		Reserved.
2	UIC interrupt active		UIC interrupt active. A status bit is asserted in the UIC status register.
1	Timer interrupt active		Timer interrupt active. A status bit is asserted in the timer status register.
0	Queue interrupt active		Queue interrupt active. A bit is asserted in the queue status register.

Table 30. Host Indirect Address Register—Address 0x30

Bit 15	Bit 14 to Bit 13	Bit 12 to Bit 8	Bit 7 to Bit 0
AINC reset	Reserved	MAC internal address	Register address

Table 31. Bit Descriptions for Indirect Address Register

Bits	Bit Name	Settings	Description
15	AINC reset		Auto-increment reset. Always reads as 0. For timer capture registers or counters, this flag can be used as the signal the capture the timer value. If the register address does not indicate a multiple access register, this bit has no effect.
		0	If the lower address is for a register that requires multiple accesses, the auto-increment value for the address is left unchanged so that reading of the overall register value continues where it was most recently started.
		1	If the lower address is for a register that requires multiple accesses (such as a 64-bit timer value), the auto-increment value for the address is set to zero.
[14:13]	Reserved		Reserved.

REGISTER MAPS AND DEFINITIONS

Table 31. Bit Descriptions for Indirect Address Register (Continued)

Bits	Bit Name	Settings	Description
[12:8]	MAC internal address		This bit is a subset of the MAC registers that are indirectly addressed within the MAC, which provides the indirect address value.
[7:0]	Register address		This bit indicates the register to be accessed on the next read or write to the indirect data registers.

Table 32. Host Indirect Read Data Register—Address 0x34

Bit 15	Bit 14 to Bit 13	Bit 12 to Bit 8	Bit 7 to Bit 0
AINC reset	Reserved	MAC internal address	Register address

Register 0x34 reads the contents that are pointed to by the host indirect address register.

Table 33. Host Indirect Write Data Register—Address 0x34

Bit 15	Bit 14 to Bit 13	Bit 12 to Bit 8	Bit 7 to Bit 0
AINC reset	Reserved	MAC internal address	Register address

Register 0x34 writes to the contents that are pointed to by the host indirect address register.

Table 34. Host Write Queue n Completion Register—Address 0x38 and Address 0x3C

Bit 31 to Bit 0
Data

When read, Register 0x38 and Register 0x3C indicate that the current write to the host controlled queue is complete, regardless of the boundary. This action causes the hardware to write out all data to memory. For example, if 32 bits of the current 64-bit memory word is written. These four bytes are written to memory. The other four bytes already in memory are left unchanged.

INDIRECT ADDRESS HOST REGISTERS

Table 35. Host Control Register, Upper Byte—Address 0x00

Bit 15 to Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Reserved	RMII clock enable	Host signal	Buffer lock	Reserved	Clock out enable	Port 2 mode

Table 36. Host Control Register, Lower Byte—Address 0x00

Bit 7	Bit 6 to Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port 2 mode	Port 1 mode	UICs ready	Port 2 signal	Port 1 signal	Programming complete (PGC)	Software reset (SWR)

Table 37. Bit Descriptions for Host Control Register

Bits	Bit Name	Settings	Description
[15:14]	Reserved		Reserved.
13	RMII clock enable		RMII clock out enable. The register write occurs immediately, but this write is not applied until the reference clock is low. Any high pulses on the reference clock are 40 ns.
		0	Disable RMII clock output.
		1	Enable 50 MHz clock output.
12	Host signal		This bit is latched in the host UIC FIFO status register. This bit is cleared by hardware.
		0	Reset.
		1	Written to 1 by the host to alert the host UIC to read a specific location in buffer space. This acts as an interrupt.
11	Buffer lock		Buffer lock.
		0	The host can write to the host space reservation register.
		1	The host cannot write to the head register.
10	Reserved		Reserved.
9	Clock out enable		Clock out enable. The register write occurs immediately, but the write is not applied until the reference clock is low. Any high pulses on the reference clock are 40 ns.
		0	Disable clock output.
		1	Enable 25 MHz clock output.
[8:5]	Port x mode		Port x interface mode.

REGISTER MAPS AND DEFINITIONS

Table 37. Bit Descriptions for Host Control Register (Continued)

Bits	Bit Name	Settings	Description
		00	10/100 MII Mbps.
		01	10/100 RMII Mbps.
		10	Not defined.
		11	Not defined.
4	UICs ready		UIC firmware loaded. This flag is asserted high when the hardware finishes loading the firmware to the respective UICs. This process begins when the host writes the PGC flag.
3	Port 2 signal		Port 2 signal. This bit is latched to the Port 2 UIC FIFO status register and is cleared by hardware.
		0	Reset.
		1	Written to 1 by the host to alert the Port 2 UIC to read a specific location in buffer space. This bit acts as an interrupt.
2	Port 1 signal		Port 1 signal. This bit is latched to the Port 1 UIC FIFO status register and is cleared by hardware.
		0	Reset.
		1	Written to 1 by the host to alert the Port 1 UIC to read a specific location in buffer space. This bit acts as an interrupt.
1	PGC		Programming complete. This bit must be set to 1 when the host finishes writing the program of the switch into memory following reset.
0	SWR		Software reset. When this bit is set to 1, the flag causes the switch to enter a reset state. The switch must be reprogrammed after this and the PGC flag must be set again.

Table 38. Protocol Register—Address 0x01

Bit 15 to Bit 7	Bit 6 to Bit 0
Protocol high value	Protocol low value

Register 0x01 is one time programmable. The exact mechanism for this programming depends on the technology chosen.

Table 39. Version Register—Address 0x02

Bit 15 to Bit 11	Bit 10 to Bit 1	Bit 0
Version	Manufacturer ID	LSB

Register 0x02 is fixed and read only.

Table 40. Part Number Register—Address 0x03

Bit 15 to Bit 0
Part Number = 0x3300

Register 0x03 is fixed and read only.

Table 41. OEM ID Register—Address 0x04

Bit 15 to Bit 0
Value

Register 0x04 has a customer specific value that allows software to be licensed or sold.

Table 42. Host Space Reservation Register—Address 0x05

Bit 15 to Bit 8	Bit 7 to Bit 0
Reserved	Reserved buffer count

Register 0x05 is used to reserve space in the internal memory of the switch for protocol specific data structures.

The reserved buffer count is the number of 256-byte buffers to reserve. The value written to this register is used to initialize the head register.

When the buffer lock flag in the host control register is set, then writes to the address registers of the host-controlled queues are limited to the space provided. Writes to the registers and auto-increment (as queues are written or read) are not allowed beyond the reserved region.

For example, suppose that the host space reservation register is written with a value of 0xA. This value reserves the range from 0x0 to 0x0AFF for use in the host interface. Following the setting of the buffer lock to 1, the maximum address accessible through the host-controlled queues (read or write) is Location 0x0AFF.

REGISTER MAPS AND DEFINITIONS

Register 0x05 must be written to a value of at least 1. This register must be written prior to enabling any host queues or Ethernet queues.

Table 43. Buffer List Stack Pointer—Address 0x06

Bit 7 to Bit 0
Stack index

The stack pointer is the address of the next element to be returned from the free list.

Table 44. Buffer List Maximum Stack Index—Address 0x07

Bit 7 to Bit 0
Maximum stack index

The maximum stack register contains the maximum value reached by the stack index register (high watermark).

Table 45. Port Configuration Register—Address 0x08

Bit 15	Bit 14	Bit 13 to Bit 9	Bit 8 to Bit 7	Bit 6 to Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port 2 link status	Port 1 link status	Reserved	Preamble mode Port 2	Preamble mode Port 1	Transmit data queue enabled	16-bit CRC check enabled	Egress/ingress timestamp enabled	Correction time management enabled	Multi-CRC check enabled

Table 46. Port CRC Offset Register—Address 0x09

Bit 7 to Bit 0
Secondary CRC offset

There is a single copy of Register 0x09 that applies to both ports. This register is accessible from the host interface.

Secondary CRC offset is the offset in bytes (plus 8 bytes for a time stamp) from the beginning of a received packet to which to compare the current CRC to 0xFFFFFFFF.

For example, if the last byte of the dual CRC is the 20th byte of data, the offset is 20 + 8 = 28, and 0x1C hex is the value placed in the offset register.

Table 47. Port n 16-Bit CRC Header Register—Address 0x0A and Address 0x0B

Bit 15 to Bit 0
Index

Register 0x0A and Register 0x0B hold the index in the packet of the location of the 16-bit CRC of the header for PROFINET.

Table 48. Port n 16-Bit CRC Start Register—Address 0x0C and Address 0x0D

Bit 15 to Bit 0
Index

Register 0x0C and Register 0x0D hold the index into the packet of the location to start a new 16-bit CRC for the final subframe for PROFINET.

Table 49. Activity LED Control Register—Address 0x0E

Bit 15 to Bit 10	Bit 9	Bit 8	Bit 7 to Bit 2	Bit 1	Bit 0
Reserved	Port 2 output	Port 2 mode	Reserved	Port 1 output	Port 1 mode

Table 50. Bit Descriptions for LED Control Register

Bits	Bit Name	Settings	Description
[15:10]	Reserved		Reserved.
9	Port 2 output		Port 2 output. Reset value of 0x0000.
		0	Output deasserted. LED off, output high.
		1	Output asserted. LED on, output low.
8	Port 2 mode		Port 2 mode.
		0	Automatic mode.
		1	Host controlled mode.

REGISTER MAPS AND DEFINITIONS

Table 50. Bit Descriptions for LED Control Register (Continued)

Bits	Bit Name	Settings	Description
[7:2]	Reserved		Reserved.
1	Port 1 output		Port 1 output.
		0	Output deasserted. LED off, output high.
		1	Output asserted. LED on, output low.
0	Port 1 mode		Port 1 mode.
		0	Automatic mode.
		1	Host controlled mode.

For a given port, if the mode is automatic, a hardware state machine controls the output state. If the mode is host controlled, the output bits of the respective port in the control register determine the state of the output. To set the behavior for the control of the LED outputs, do the following:

- When activity is present on a port (transmit or receive of packets), the LED is driven on.
- When no activity is present on a port, the LED is maintained in the off state.
- On time is no less than 100 ms.
- Activity is any transmit or receive activity on the given port.

The implementation of the blinking is as follows:

1. A 24-bit down counter clocked by the internal 125 MHz clock is used for control. If the counter is nonzero, it decrements.
2. If the down counter is 0, the LED output is disabled (high).
3. If the down counter is nonzero, the LED output is enabled (low).
4. On the rising edge of Pin P1_RXDV or Pin P2_RXDV on the fido5100 and fido5200, the counter is loaded to full scale (0xFFFFFF).
5. On the rising edge of Pin P1_TXEN or Pin P2_TXEN on the fido5100 and fido5200, the counter is loaded to full scale (0xFFFFFF).

The result is an LED output that is asserted for 2^{24} × 8 ns (134 ms) on the detection of any transmit or receive activity on a port. If activity is continuous, the LED is on continuously.

Table 51. Host Write Queue 0 Address—Address 0x18

Bit 15 to Bit 1	Bit 0
Write	0

Table 52. Host Read Queue 0 Address—Address 0x19

Bit 15 to Bit 1	Bit 0
Read address	0

The LSB of the address register is one byte and all valid addresses are word aligned. Bit 0 is used to indicate that the data is ready to be read.

Table 53. Bit Descriptions for Host Read Queue 0 Address Register

Bits	Bit Name	Settings	Description
[15:1]	Read address		Contains the address to be read, assuming the LSB is 0.
0	0		The value of this bit determines whether there is valid data in the read register.
		0	The valid data is present in the read register.
		1	The valid data is not present in the read register. The host can poll this flag prior to beginning a read.

Table 54. Host Write Queue 1 Address—Address 0x1A

Bit 31 to Bit 16	Bit 15 to Bit 2	Bit 1	Bit 0
Reserved	Write address	0	0

All accesses on Register 0x1A are limited to the bus width. When the host writes this register, the write queue is flushed. The next write takes place to this address, then subsequent writes to the queue go to the write addresses that were just passed in. The current address of the queue is what is returned on a read from this register.

REGISTER MAPS AND DEFINITIONS

Table 55. Host Read Queue 1 Address—Address 0x1B

Bit 31 to Bit 16	Bit 15 to Bit 2	Bit 1	Bit 0
Reserved	Read address	0	0

Two LSBs of the register are always set to zero, and all reads are long word aligned.

Table 56. Bit Descriptions for Host Read Queue 1 Address Register

Bits	Bit Name	Settings	Description
[31:16]	Reserved		Reserved.
[15:2]	Read address		Contains the address to be read, assuming the LSB is 0.
[1:0]	0		The value of this bit determines whether there is valid data in the read register.
		0	The valid data is present in the read register.
		1	The valid data is not present in the read register. The host can poll this flag prior to beginning a read.

Table 57. Host Read Queue Control Register, Upper Byte—Address 0x1C

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Enable Queue 3	Flush Queue 3	FCP 3	QNE 3	Enable Queue 2	Flush Queue 2	FCP 2	QNE 2

Table 58. Host Read Queue Control Register, Lower Byte—Address 0x1C

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable Queue 1	Flush Queue 1	FCP 1	QNE 1	Enable Queue 0	Flush Queue 0	FCP 0	QNE 0

Table 59. Bit Descriptions for Host Read Queue 1 Address Register

Bits	Bit Name	Settings	Description
15, 11, 7, 3	Enable Queue x		Enable queue.
		0	Hardware queue is operational.
		1	Hardware puts all registers in a reset state.
14, 10, 6, 2	Flush Queue x		When these bits are set to 1, the hardware treats the current packet read as if it completed and drops any pending packets. This signal is forwarded to the UIC flush register. These bits are cleared by hardware once flush is complete.
13, 9, 5, 1	FCP x		Flush current packet. When these bits are set to 1, the hardware treats the current packet read as if it completed and moves on to the next message if present. If the end of a packet is currently in the queue and a second one started, the last data of the packet in the queue is lost. The second packet is also marked as complete and flushed. These bits are cleared by hardware.
12, 8, 4, 0	QNE x		Queue not empty. Same as flag in read queue status register for this queue. This reflects the actual state, the bit in Q status is an interrupt flag that is set on transition and cleared by the host.

Table 60. Host Write Queue Control Register, Upper Byte—Address 0x1E

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Enable Queue 3	Flush Queue 3	FCP 3	Queue Ready 3	Enable Queue 2	Flush Queue 2	FCP 2	Queue Ready 2

Table 61. Host Write Queue Control Register, Lower Byte—Address 0x1E

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Enable Queue 1	Flush Queue 1	FCP 1	Queue Ready 1	Enable Queue 0	Flush Queue 0	FCP 0	Queue Ready 0

Table 62. Bit Descriptions for Host Write Queue Control Register

Bits	Bit Name	Settings	Description
15, 11, 7, 3	Enable Queue x		Enable queue.
14, 10, 6, 2	Flush Queue x		Flush queue.
		0	No action.
		1	Discards all packets written on the queue and any in progress. Cleared by hardware.
13, 9, 5, 1	FCP x		Flush current packet.
		0	No action.
		1	Discontinues the current write packet and frees the used buffers. Cleared by hardware.
12, 8, 4, 0	Queue Ready x		These bits are a copy of the same flag in the write queue status register.

REGISTER MAPS AND DEFINITIONS

Table 63. Atomic Communication Register n—Address 0x20, Address 0x21, Address 0x22, Address 0x23, Address 0x24, Address 0x25, Address 0x26, and Address 0x27

Bit 15 to Bit 8	Bit 7 to Bit 0
Reserved	Data

There is a single machine register address for writes to all eight copies of the atomic registers. The specific register impacted is determined by the contents of the atomic index register. The host atomic read data register contains the data that was in the atomic write data register at the time that the host interface last wrote to that register.

There is a single machine register address for reads from all eight copies of atomic registers. The specific register impacted is determined by the contents of the atomic index register.

Table 64. Atomic Writer Status Register—Address 0x28

Bit 15 to Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Register 7 status	Register 6 status	Register 5 status	Register 4 status	Register 3 status	Register 2 status	Register 1 status	Register 0 status

Table 65. Bit Descriptions for Atomic Writer Status Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:0]	Register x status		Register x Status. These bits determine which interface most recently wrote to the atomic read data register x.
		0	The host UIC was the last writer.
		1	The host interface was the last writer.

Table 66. Queue Interrupt Mask Register—Address 0x30

Bit 15 to Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7 to Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Queue 3 ready	Queue 2 ready	Queue 1 ready	Queue 0 ready	Reserved	Queue 3 packet ready	Queue 2 packet ready	Queue 1 packet ready	Queue 0 packet ready

Table 67. Bit Descriptions for Queue Interrupt Mask Register

Bits	Bit Name	Settings	Description
[15:12]	Reserved		Reserved.
[11:8]	Queue x ready		Queue x ready.
		0	Disable interrupt.
		1	Enable interrupt.
[7:4]	Reserved		Reserved.
[11:8]	Queue x packet ready		Queue x packet ready.
		0	Disable interrupt.
		1	Enable interrupt.

If the bus interface is 16 bits, then the register is returned as shown. If the bus interface is 32 bits, then this is returned as the lower 16 bits of the data bus.

Table 68. Timer Interrupt Setup Register—Address 0x32

Bit 15 to Bit 14	Bit 13 to Bit 12	Bit 11 to Bit 10	Bit 9 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Compare 3	Compare 2	Compare 1	Compare 0	Capture 3	Capture 2	Capture 1	Capture 0

Table 69. Bit Descriptions for Timer Interrupt Setup Register

Bits	Bit Name	Settings	Description
[15:8]	Compare x		Compare.
		00	Use Interrupt 0.
		01	Use Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.
[7:0]	Capture x		Capture.

REGISTER MAPS AND DEFINITIONS

Table 69. Bit Descriptions for Timer Interrupt Setup Register (Continued)

Bits	Bit Name	Settings	Description
		00	Use Interrupt 0.
		01	Use Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.

Table 70. Timer Control Unit Interrupt Setup Register—Address 0x33

Bit 15 to Bit 14	Bit 13 to Bit 12	Bit 11 to Bit 10	Bit 9 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Reserved	TCU Interrupt 2	TCU Interrupt 1	TCU Interrupt 0	TCU 3	TCU 2	TCU 1	TCU 0

Table 71. Bit Descriptions for Timer Control Unit Interrupt Setup Register

Bits	Bit Name	Settings	Description
[15:14]	Reserved		Reserved.
[13:8]	TCU Interrupt x		TCU Interrupt x.
[7:0]	TCU x		TCU x.
		00	Use Interrupt 0.
		01	Use Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.

Table 72. Timer Interrupt Mask Register—Address 0x34

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7 to Bit 4	Bit 3 to Bit 0
Reserved	TCU Interrupt 2	TCU Interrupt 1	TCU Interrupt 0	TCU 3	TCU 2	TCU 1	TCU 0	Compare	Capture

Table 73. Bit Descriptions for Timer Interrupt Mask Register

Bits	Bit Name	Settings	Description
15	Reserved		Reserved.
[14:12]	TCU Interrupt x		TCU Interrupt x.
		0	Disable interrupt.
		1	Enable interrupt.
[11:8]	TCU x		TCU x.
[7:4]	Compare		Compare.
[3:0]	Capture		Capture.

Table 74. Link Status Interrupt Setup Register—Address 0x35

Bit 15 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Reserved	Timer Periodic Interrupt 1	Timer Periodic Interrupt 0	Port 2 link status interrupt	Port 1 link status interrupt

Table 75. Bit Descriptions for Link Status Interrupt Status Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:4]	Timer Periodic Interrupt x		Timer Periodic Interrupt x.
[3:0]	Port x link status interrupt		Port x link status interrupt.
		00	Use Interrupt 0.
		01	User Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.

Table 76. UIC Interrupt Setup Register—Address 0x36

Bit 15 to Bit 14	Bit 13 to Bit 12	Bit 11 to Bit 10	Bit 9 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Host UIC Interrupt 3	Host UIC Interrupt 2	Host UIC Interrupt 1	Host UIC Interrupt 0	Port 2 UIC Interrupt 1	Port 2 UIC Interrupt 0	Port 1 UIC Interrupt 1	Port 1 UIC Interrupt 0

REGISTER MAPS AND DEFINITIONS

Table 77. Bit Descriptions for Link Status Interrupt Status Register

Bits	Bit Name	Settings	Description
[15:8]	Host UIC Interrupt x		Host UIC Interrupt x.
		00	Use Interrupt 0.
		01	Use Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.
[7:4]	Port 2 UIC Interrupt x		Port 2 UIC Interrupt x.
[3:0]	Port 1 UIC Interrupt x		Port 1 UIC Interrupt x.

Table 78. UIC Interrupt Mask Register, Upper Byte—Address 0x37

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11 to Bit 8
Port 2 link status change	Port 1 link status change	Timer Periodic Interrupt 1	Timer Periodic Interrupt 0	Reserved

Table 79. UIC Interrupt Mask Register, Lower Byte—Address 0x37

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Host UIC Interrupt 3	Host UIC Interrupt 2	Host UIC Interrupt 1	Host UIC Interrupt 0	Port 2 UIC Interrupt 1	Port 2 UIC Interrupt 0	Port 1 UIC Interrupt 1	Port 1 UIC Interrupt 0

Table 80. Bit Descriptions for UIC Interrupt Mask Register

Bits	Bit Name	Settings	Description
[15:14]	Port x link status change		Port x link status change.
[13:12]	Timer Periodic Interrupt x		Timer periodic Interrupt x.
[11:8]	Reserved		Reserved.
[7:4]	Host UIC Interrupt x		Host UIC Interrupt x.
		0	Disable interrupt.
		1	Enable interrupt.
[3:2]	Port 2 UIC Interrupt x		Port 2 UIC Interrupt x.
[1:0]	Port 1 UIC Interrupt x		Port 1 UIC Interrupt x.

Table 81. Host Queue Packet Ready Interrupt Setup Register—Address 0x3A

Bit 15 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Reserved	Q3	Q2	Q1	Q0

Table 82. Bit Descriptions for Host Queue Packet Ready Interrupt Setup Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:0]	Qx		Queue x.
		00	Use Interrupt 0.
		01	Use Interrupt 1.
		10	Use Interrupt 2.
		11	Reserved.

Table 83. Host Queue Space Available Interrupt Setup Register—Address 0x3B

Bit 15 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Reserved	Q3	Q2	Q1	Q0

Table 84. Bit Descriptions for Host Queue Space Available Interrupt Setup Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:0]	Qx		Queue x.
		00	Use Interrupt 0.
		01	Use Interrupt 1.

REGISTER MAPS AND DEFINITIONS

Table 84. Bit Descriptions for Host Queue Space Available Interrupt Setup Register (Continued)

Bits	Bit Name	Settings	Description
		10	Use Interrupt 2.
		11	Reserved.

Table 85. TCU Control Register—Address 0x40

Bit 15 to Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Active B	Active A	Enable B	Enable A

Table 86. Bit Descriptions for TCU Control Register

Bits	Bit Name	Settings	Description
[15:4]	Reserved		Reserved.
[3:2]	Active x		Active x.
		0	Program x is not executing.
		1	Program x is currently being executed.
[1:0]	Enable x		Enable x is the enable flag for TCU Program x.
		0	Not enabled, ready to program.
		1	Enabled, ready to execute but cannot program.

Table 87. Timer Register n—Address 0x41

Bit 63 to Bit 32	Bit 31 to Bit 4	Bit 3 to Bit 0
Programmable portion of the timer	Nanoseconds	Reserved

The host access Register 0x41 as 64 bits. During a write by the host, the lower 4 bits are discarded. The lower 4 bits are represented by the upper 4 bits of the timer accumulator register.

If the timer addend register (Timer Addend Register A by default, but the user can select Timer Addend Register B) value overflows when the register is written to, the overflow bits are added into the Field Encompassing Bits[31:4]. When Bits[31:4] equal the value in the corresponding bits of the timer update compare register, those bits are cleared, and the Field Encompassing Bits[63:32] increment.

Timer Addend Register A is used if Timer Addend Register B count is zero. Otherwise, Timer Addend Register B is used.

All fields are set to zero at reset.

The host accesses this register as a series of 16-bit registers. For a write, the value is not applied until all four registers are written to. For a read, when the lowest register (Bits[15:4]) is read, all register values are latched for subsequent reads.

Table 88. Bit Descriptions for Timer Register n

Bits	Bit Name	Settings	Description
[63:32]	Programmable portion of the timer		Programmable portion of the timer. These bits represent the programmable portion of the time (Bit 32 is the value, in ns, of the timer update compare register).
[31:4]	Nanoseconds		These bits represent the nanoseconds portion of the time (Bit 4 = 8 ns, Bit 28 of the accumulator = 1 ns).
[3:0]	Reserved		Reserved.

Table 89. Timer Addend Register A—Address 0x42

Bit 31 to Bit 16	Bit 15 to Bit 0
Addend value (Auto-Increment 1)	Addend value (Auto-Increment 0)

Table 90. Timer Addend Register B Count—Address 0x43

Bit 31 to Bit 16	Bit 15 to Bit 0
Addend value (Auto-Increment 1)	Addend value (Auto-Increment 0)

Register 0x42 and Register 0x43 have a 32-bit value. These registers are decremented each time Timer Addend Register B is added to the timer accumulator register. When the value reaches zero, Timer Addend Register A updates the timer accumulator register. These registers are only accessible by the host.

REGISTER MAPS AND DEFINITIONS

Table 91. Timer Addend Register B—Address 0x44

Bit 31 to Bit 16	Bit 15 to Bit 0
Addend value (Auto-Increment 1)	Addend value (Auto-Increment 0)

If the Timer Addend Register B count is nonzero, the addend value is added with each system clock to the timer accumulator register, instead of Timer Addend Register A. The value of this register represents the time of a single system clock. This register is only accessible by the host, and the host adjusts the register value to slow down or speed up the 1588 ns clock. This register is set to 80000000h at reset (8 ns for a 125 MHz system clock).

Table 92. Timer I/O Control Register—Address 0x45

Bit 15 to Bit 8	Bit 7 to Bit 6	Bit 5 to Bit 4	Bit 3 to Bit 2	Bit 1 to Bit 0
Reserved	Timer 3	Timer 2	Timer 1	Timer 0

Table 93. Bit Descriptions for Timer I/O Control Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:0]	Timer x		Timer x is associated with Timer Output Compare Register x and Timer Input Capture Register x.
		00	Input capture register. Pin is configured as an input, negative edge triggered.
		01	Input capture register. Pin is configured as an input, positive edge triggered.
		10	Output compare register. Pin is configured as an output.
		11	Pin is high impedance, not monitored.

Table 94. TCU Start Compare Register—Address 0x46

Bit 31 to Bit 4	Bit 3 to Bit 0
TCU start time (in nanoseconds)	Reserved

Writing to Register 0x46 by a pair of writes from the host enables a comparator with the lower 32 bits of the Register 0x41, which ignores the bottom 4 bits. When the comparison is successful, a signal is generated to start the TCU and the machine is disabled until the next write to the value register by the host.

Table 95. Port n Delay Registers—Address 0x47 and Address 0x48

Bit 15 to Bit 0
Port transmit delay

Register 0x47 and Register 0x48 hold a 16-bit value that can be loaded to a decrementer by a write from the opposite port UIC. As such, the Port 1 UIC loads the Port 2 decrementer, and the Port 2 UIC loads the Port 1 decrementer. When the decrementer contains a nonzero value, it decrements at the same rate that as the Register 0x41. A 1 is sent to the associated port transmit state machine when the value in the decrementer is zero.

Receiving UIC sets a bit in a machine register that loads the decrementer when a packet is in the process of being received from the port. On the transmit side, the transmitting UIC determines on a per packet basis, whether the port pays attention to the decrementer or not. The transmit port hardware begins transmitting preamble when the flag is clear, or when the flag is high and the decrementer reaches zero. If a packet ID is provided to the port transmit hardware, the decrementer signal is low, and there was no previous transition to start preamble transmission, the hardware begins transmission.

Table 96. Timer Periodic Interrupt n Registers—Address 0x49 and Address 0x4A

Bit 31 to Bit 4	Bit 3 to Bit 0
Timer reload value	Reserved

When a nonzero value is written to Register 0x49 and Register 0x4A, the value is copied to a register that decrements whenever the accumulator of the Register 0x41 overflows. When the decrementer reaches zero, an interrupt is generated and the value of this register is reloaded. No interrupt is generated if this register is zero.

Table 97. Port 1 Egress Time Register B x—Address 0x4B

Bit 63 to Bit 32	Bit 31 to Bit 0
Seconds	Nanoseconds

REGISTER MAPS AND DEFINITIONS

Register 0x4B contains the egress time of a packet.

Table 98. Port 2 Egress Time Register B x—Address 0x4C

Bit 63 to Bit 32	Bit 31 to Bit 0
Seconds	Nanoseconds

Table 99. TCU Memory Register x—Address 0x4D and Address 0x4E

Bit 15 to Bit 0
Write Data

Register 0x4D and Register 0x4E are used to initialize the corresponding program table in the TCU. Each write to these registers loads 16 bits to the program table. After each write, the write location is incremented by 1. Each entry in the table requires three writes. After the table is enabled in the TCU control register, writes to these registers have no effect.

Table 100. TCU Error Addend Register—Address 0x4F

Bit 15 to Bit 8	Bit 7 to Bit 0
Bank B addend	Bank A addend

Register 0x4F contains two fields, one for each TCU program memory. The contents of the memory are used to calculate the current error of the TCU program with respect to the target time so that clocks can be skipped to eliminate the error. Bank A addend and Bank B addend determine the value of the error to be added to the accumulator and the end of each pass of Program A and Program B, respectively.

Table 101. Timer Input Capture Register A x—Address 0x50

Bit 63 to Bit 0
Compare value

Each timer input capture register can be mapped to a unique timer input. Each register can be configured to trigger on either positive or negative edge events on the timer pin. The value in the Register 0x41 is stored in Register 0x50 and a flag is set in the timer status register. Register 0x50 is accessible by the host.

Table 102. Timer Input Capture Register B x—Address 0x51

Bit 63 to Bit 0
Capture value

Table 103. Timer Input Capture Register C x—Address 0x52

Bit 63 to Bit 0
Capture value

Table 104. Timer Input Capture Register D x—Address 0x53

Bit 63 to Bit 0
Capture value

Table 105. Timer Output Compare Register A x—Addresses 0x54

Bit 63 to Bit 4	Bit 3 to Bit 0
Compare value	Reserved

The value in Register 0x54 is compared to the Register 0x41 and sets a flag in the timer status register if there is a match between the two times. Each timer output compare register can be configured to generate a signal on one of the timer pins on a match. Register 0x50 to Register 0x57 are accessible by the host.

Table 106. Timer Output Compare Register B x—Address 0x55

Bit 63 to Bit 4	Bit 3 to Bit 0
Compare value	Reserved

Table 107. Timer Output Compare Register C x—Address 0x56

Bit 63 to Bit 4	Bit 3 to Bit 0
Compare value	Reserved

REGISTER MAPS AND DEFINITIONS

Table 108. Timer Output Compare Register D x—Address 0x57

Bit 63 to Bit 4	Bit 3 to Bit 0
Compare value	Reserved

Table 109. Port 1 Egress Time Register A x—Address 0x58

Bit 63 to Bit 32	Bit 31 to Bit 0
Seconds	Nanoseconds

Table 110. Port 2 Egress Time Register A x—Address 0x59

Bit 63 to Bit 32	Bit 31 to Bit 0
Seconds	Nanoseconds

Table 111. Port 1 Peer Delay Register 0—Address 0x5A

Bit 15 to Bit 0
Peer delay, Bits[15:0]

There is one set of peer delay registers for each port.

Table 112. Port 1 Peer Delay Register 1—Address 0x5A

Bit 31 to Bit 16
Peer delay, Bits[31:16]

Table 113. Port 1 Peer Delay Register 2—Address 0x5A

Bit 47 to Bit 32
Peer delay, Bits[47:32]

Table 114. Port 1 Peer Delay Register 3—Address 0x5A

Bit 63 to Bit 48
Peer delay, Bits[63:48]

Table 115. Port 2 Peer Delay Register 0—Address 0x5B

Bit 15 to Bit 0
Peer delay, Bits[15:0]

Table 116. Port 2 Peer Delay Register 1—Address 0x5B

Bit 31 to Bit 16
Peer delay, Bits[31:16]

Table 117. Port 2 Peer Delay Register 2—Address 0x5B

Bit 47 to Bit 32
Peer delay, Bits[47:32]

Table 118. Port 2 Peer Delay Register 3—Address 0x5B

Bit 63 to Bit 48
Peer delay, Bits[63:48]

Table 119. Timer Update Compare Register—Address 0x5C

Bit 31 to Bit 16	Bit 15 to Bit 5	Bit 4 to Bit 1
Compare value, Bits[31:16]	Compare value, Bits[15:0]	Ignored

The compare value is the value is used for comparison so that the lower 32 bits of the timer are reset and the upper 32 bits are incremented. Ignored indicates the portion of the register correlates with the upper bits of the accumulator. The upper 15 bits and lower 15 bits of this register have different reset values, as noted in Table 13.

REGISTER MAPS AND DEFINITIONS

Table 120. Address Lookup Control Register—Address 0x60

Bit 15 to Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Write protect enable	Register write protect enable	FMMU program enable	Enable

Table 121. Maximum Valid Address Register—Address 0x61

Bit 15 to Bit 0
Maximum address

Table 122. Write Offset Register—Address 0x62

Bit 15 to Bit 0
Write offset

Write offset is the 16-bit unsigned value indicating the difference between the read address (provided in PDU) and write address for specific instructions, which only applies to configured address physical read/writer (FPRW) and auto-increment physical read/write (APRW) commands.

Table 123. FMMU List Programming Register—Address 0x63

Bit 15 to Bit 0
Table data

The FMMU list is maintained in RAM and has eight entries of the following format:

- Logical start address is a 32-bit logical address for the start of the block covered by the FMMU, Bits[31:0].
- Logical end address is a 32-bit logical address for the end of the block covered by the FMMU, Bits[31:0].
- Physical address is 16-bit unsigned local address. This address corresponds to the value in the logical address, Bits[15:0].
- Control is described in detail in Table 124 (see Bits[15:0]).

Table 124. Control

Bit 15 to Bit 8	Bit 7 to Bit 3	Bit 2	Bit 1	Bit 0
AND mask	Reserved	Mask enable	Write enable	Read enable

Table 125. Bit Descriptions for Control Register

Bits	Bit Name	Settings	Description
[15:8]	AND mask		AND mask.
[7:3]	Reserved		Reserved.
2	Mask enable		Mask enable is used to emulate the shift function.
		0	Normal operation for FMMU.
		1	Read the value from the physical address and AND the value with the value of the AND mask, AND incoming data with the inverse of the AND mask, then OR the two results together. Only applies to read FMMU.
1	Write enable		Write enable.
		0	Does not define translation for write operations.
		1	Use for write translations.
0	Read enable		Read enable. If neither write enable or read enable is asserted, the entry is not valid.
		0	Does not define translation for read operations.
		1	Use for read translations.

The FMMU list is programmed through a sequence of writes to a programming register, as shown in Table 126. This sequence of writes is repeated eight times to fill all eight lines in the table. There is no way to program a portion of the table, and every line must have an entry.

Table 126. FMMU List Programming

Write 0	Write 1	Write 2	Write 3	Write 4	Write 5
Logical address, Bits[31:16]	Logical address, Bits[15:0]	End address, Bits[31:16]	End address, Bits[15:0]	Physical start address[15:0]	Control

REGISTER MAPS AND DEFINITIONS

Table 127. SyncManager Programming Register—Address 0x64

Bit 15 to Bit 0
Program data

In Register 0x64, each SyncManager is programmed by a sequence of five sequential writes, which are described in Table 128. Excess writes to the SyncManager programming register for a given write to the SyncManager program index register are discarded.

Table 128. SyncManager List

Start Address		Buffer 1 Start Address		Buffer 2 Start Address		End Address		Control	
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9

The start address for the SyncManager is a 16-bit physical address where the SyncManager operates.

The Buffer 1 start address is a 16-bit physical address where Buffer 1 (second buffer) begins. This register must be set to end address + 1 if the SyncManager is in mailbox mode.

The Buffer 2 start address is a 16-bit physical address where Buffer 2 (third buffer) begins. This register must be set to end address + 1 if the SyncManager is in mailbox mode.

The end address is the last address managed by the SyncManager. For mailbox mode, the end address is equal to the start address plus the length of the SyncManager – 1. For buffered mode, the end address is equal to the Buffer 2 start address plus the length of the SyncManager − 1.

The control functions are described in Table 129.

Table 129. SyncManager List Control Field

Bit 15 to Bit 2	Bit 1	Bit 0
Reserved	Mode	Direction

Table 130. Bit Descriptions for SyncManager List Control Field

Bits	Bit Name	Settings	Description
[15:2]	Reserved		Reserved.
1	Mode		Mode.
		0	Buffered SyncManager.
		1	Mailbox SyncManager.
0	Direction		Direction.
		0	Read SyncManager.
		1	Write SyncManager.

The set of values in Table 130 representing a single SyncManager can be programmed using the SyncManager programming register and the SyncManager program index register, as follows:

- Use the SyncManager control/state register to disable the SyncManager that is to be programmed.
- Write the index of the SyncManager to the SyncManager program index register, Bits[0:7], to set the machine that manages writes to the SyncManager programming register to point to the first word.
- Write the sequence of words described in Table 131 for that SyncManager.
- Write the appropriate value to the SyncManager control/state register to enable the SyncManager.

When the synchronization manger is being programmed, all other SyncManagers are operational.

Table 131. SyncManager Control/State Register—Address 0x65

Bit 15 to Bit 8	Bit 7	Bit 6	Bit 5 to Bit 2	Bit 1	Bit 0
Index	Enable	Restricted	Reserved	Full	Open

Table 132. Bit Descriptions for SyncManager Control/State Register

Bits	Bit Name	Settings	Description
[15:8]	Index		Index indicates the specific register (between 0 and 7) to which to apply the write.

REGISTER MAPS AND DEFINITIONS

Table 132. Bit Descriptions for SyncManager Control/State Register (Continued)

Bits	Bit Name	Settings	Description
7	Enable		Enable allows a write from host. Note that a transition of this flag (from 0 to 1) initializes the SyncManager (current buffer is none, open is 0, full is 0).
		0	SyncManager is not active, do not use in address evaluation.
		1	SyncManager is active.
6	Restricted		Restricted allows a write from the host. A transition of this flag (from 0 to 1) initializes the SyncManager (current buffer is none, open is 0, full is 0).
		0	SyncManager operates normally.
		1	Any access to SyncManager space is restricted.
[5:2]	Reserved		Reserved.
1	Full		Full allows a host write either direction, and can be set and cleared by hardware.
		0	Buffer is not full. If open, can be written in a write SyncManager (otherwise restricted).
		1	Buffer is full. If open, can be read in a read SyncManager (otherwise restricted).
0	Open		Open is clear only from the host. Ensure that this bit never writes 1.
		0	First byte of buffer not accessed, access to remainder of buffer is restricted.
		1	First byte of buffer has been accessed, access to remainder of the buffer is unrestricted.

Table 133. SyncManager Program Index Register—Address 0x66

Bit 15 to Bit 3	Bit 2 to Bit 0
Reserved	Index

Register 0x66 contains the index of the SyncManager, which is used to program the set of writes in Register 0x68 and Register 0x6F to the SyncManager programming register.

Table 134. SyncManager Atomic Status Register—Address 0x67

Bit 15 to Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Atomic Status 7	Atomic Status 6	Atomic Status 5	Atomic Status 4	Atomic Status 3	Atomic Status 2	Atomic Status 1	Atomic Status 0

Table 135. Bit Descriptions for SyncManager Atomic Status Register

Bits	Bit Name	Settings	Description
[15:8]	Reserved		Reserved.
[7:0]	Atomic Status x		Atomic Status x indicates which entry was last written to the atomic register.
		0	Lookup engine was last writer.
		1	Host interface was last writer.

Table 136. SyncManager x Atomic Read/Write Registers—Address 0x68, Address 0x69, Address 0x6A, Address 0x6B, Address 0x6C, Address 0x6D, Address 0x6E, and Address 0x6F

Bit 15 to Bit 2	Bit 1 to Bit 0
Reserved	Buffer index

The buffer index is the ID (0 to 2) of the buffer to provide to the lookup engine. The buffer index also indicates the value that was in the atomic register the last time that the SyncManager write register was written to.

Table 137. Interframe Gap Set Registers—Address 0x80 and Address 0xB0

Bit 15 to Bit 0
Gap size

Register 0x80 and Register 0xB0 configure the size of the transmit interframe gap. The Ethernet specification calls a minimum gap 12 byte times. The register value is in byte times.

Table 138. Full Duplex Control Registers—Address 0x81 and Address 0xB1

Bit 15 to Bit 1	Bit 0
Reserved	Full duplex

REGISTER MAPS AND DEFINITIONS

Register 0x81 and Register 0xB1 are used only in 10 Mbps or 100 Mbps mode. When the register is set, the transmit state works in full duplex mode. When the register is cleared, the transmit state works in half-duplex mode. These registers detect collisions, perform random back offs, retransmit collision packets, and perform other duplex operations.

Table 139. Bit Descriptions for Full Duplex Control Register

Bits	Bit Name	Settings	Description
[15:1]	Reserved		Reserved.
0	Full duplex		Full duplex.
		0	Full duplex operation disabled.
		1	Full duplex operation enabled.

Table 140. Maximum Retry Registers—Address 0x82 and Address 0xB2

Bit 15 to Bit 5	Bit 4 to Bit 0
Reserved	Maximum retry

When the REM switch is operating in half-duplex mode, this register contains the maximum number of times the transmit logic attempts to retransmit a packet after a collision. If this number is reached, the packet is dropped.

Table 141. CRC Check Enable Registers—Address 0x84 and Address 0xB4

Bit 15 to Bit 1	Bit 0
Reserved	Enable

Table 142. Bit Descriptions for CRC Check Enable Register

Bits	Bit Name	Settings	Description
[15:1]	Reserved		Reserved.
0	Enable		This bit is the global enable to the block. It enables the FMMU program, register write protect, and write protect.
		0	Block does not operate. For FMMU program enable, the block is in normal operating mode and writes to programming registers are ignored. For register write protect, no write protection is in place. For write protect, no register protection is in place.
		1	Block is operational. For FMMU program enable, the block is in programming mode and returns miss for all accesses from the UIC. For register write protect, any register writes must be protected by a write to the write protect override register (Address 0x0020). For write protect, any register or memory writes must be proceeded by a write to the write protect override register (Address 0x0030).

Table 143. Receive Interframe Gap Control Registers—Address 0x85 and Address 0xB5

Bit 15 to Bit 0
Gap size

Register 0x85 and Register 0xB5 configure the size of the receive interframe gap. The register value is in byte times.

Table 144. Maximum Receive Length Registers—Address 0x86 and Address 0xB6

Bit 15 to Bit 0
Minimum receive packet length (bytes)

Register 0x86 and Register 0xB6 set the minimum length of an incoming packet (LSB = 1 byte). If a shorter packet is received, the receive UIC signals that an error occurred in the packet.

Table 145. Minimum Receive Length Registers—Address 0x87 and Address 0xB7

Bit 15 to Bit 0
Minimum receive packet length (bytes)

Register 0x87 and Register 0xB7 set the minimum length of an incoming packet (LSB = 1 byte). If a shorter packet is received, the receive UIC signals that an error occurred in the packet.

Table 146. Statistic Register Read Registers—Address 0x89 and Address 0xB9

Bit 31 to Bit 16	Bit 15 to Bit 0
Most significant bits	Least significant bits

REGISTER MAPS AND DEFINITIONS

Register 0x89 and Register 0xB9 return the most significant and least significant 16 bits of a 32-bit MAC statistics register. The register selection is based on the MAC address placed in the indirect address register. Possible MAC addresses are listed in Table 147.

Table 147. MAC Addresses for Register Selection

MAC Internal Address	Statistic Counter
5'b00000	rx_pkt_cnt_port
5'b00001	rx_pkt_cnt
5'b00010	rx_broadcst_cnt
5'b00011	rx_mulitcast_cnt
5'b00100	rx_unicast_cnt
5'b00101	rx_crc_err_cnt
5'b00110	rx_alignment_err_cnt
5'b00111	rx_long_short_err_cnt
5'b01000	Unused
5'b01001	Unused
5'b01010	Unused
5'b01011	Unused
5'b01100	Unused
5'b01101	Unused
5'b01110	Unused
5'b01111	rx_pkt_gt_1522
5'b10000	tx_pkt_cnt_port
5'b10001	tx_pkt_cnt
5'b10010	tx_broadcst_cnt
5'b10011	tx_mulitcast_cnt
5'b10100	tx_unicast_cnt
5'b10101	tx_jam_drop_cnt
5'b10110	tx_crc_err_cnt
5'b10111	Unused
5'b11000	tx_alignment_err_cnt
5'b11001	tx_single_collision_cnt
5'b11010	tx_multiple_collision_cnt
5'b11011	tx_deferred_transmission_cnt
5'b11100	tx_late_collision_cnt
5'b11101	tx_carrier_sense_err_cnt
5'b11110	Unused
5'b11111	Unused

Table 148. Speed Registers—Address 0x8C and Address 0xBC

Bit 15 to Bit 3	Bit 2 to Bit 0
Reserved	Speed

Table 149. Bit Descriptions for Speed Register

Bits	Bit Name	Settings	Description
[15:3]	Reserved		Reserved.
[2:0]	Speed		Speed.
		001	Speed is set to 10 Mbps.
		010	Speed is set to 100 Mbps.
		100	Speed is set to 1000 Mbps.

FUNCTION REENTRANCY

This section discusses of the reentrancy of functions within the fido5100 and fido5200 drivers. This section contains functions that are specific to a given industrial Ethernet protocol, as well as functions that are applicable to all use cases of the fido5100 and fido5200 drivers.

Table 150. General REM Switch Driver Function Reentrancy

Function	Reentrant	Notes
REMS_StdInit	Not applicable	This function is called once at initialization.
REMS_StdSetMacAddress	Not applicable	This function is called once at initialization.
REMS_StdSetSecMacAddress	Not applicable	This function is called once at initialization.
REMS_StdSetSpeedAndDuplex	Not applicable	This function is called once at initialization.
REMS_StdAssignInterrupt	Not applicable	This function is called once at initialization.
REMS_StdManageLED	Not applicable	This function is called once at initialization.
REMS_StdEvaluateInterrupt	Not applicable	This function is only called by interrupt handlers.
REMS_StdGetNextEvent	Not applicable	This function is only called by interrupt handlers.
REMS_StdGetLinkState	Yes	
REMS_StdSetPortState	No	This function writes to switch memory.
REMS_StdEnableInterrupt	No	This function modifies global data.
REMS_StdDisableInterrupt	No	This function modifies global data.
REMS_stdPollInterrupt	Not applicable	This function does not exist.
REMS_StdPollReceiveQueue	Yes	
REMS_StdXmitPacket	No	This function writes to Switch Queue 0.
REMS_StdXmitTaggedPacket	No	This function writes to Switch Queue 0.
REMS_StdReadPacket	No	This function reads from Switch Queue 0.
REMS_StdReadMacStatistic	No	Reading the switch statistic register clears the register.
REMS_StdSetTimerIO	Yes	
REMS_StdSetTimerOutputCompare	Yes	
REMS_StdGetTimerInputCapture	Yes	
REMS_StdReadTimer	No	Register 0x41 sequencer is reset within this function.
REMS_StdGetProtocolReg	Yes	

Table 151. EtherNet/IP Driver Function Reentrancy

Function	Reentrant	Notes
REMS_SetPhyDelayValues	Yes	
REMS_EipSetQueue0filterCount	No	This function writes to switch memory.
REMS_EipSetDSCPValues	No	This function writes to switch memory.
REMS_EipSetFilterCounters	No	This function writes to switch memory.
REMS_EipServiceBcastMcastFilter	Yes	
REMS_DlrXmitPacket	No	This switch writes to Switch Queue 2.
REMS_DlrReadPacket	No	This function reads from Switch Queue 2.
REMS_Class1XmitPacket	No	This function writes to Switch Queue 1.
REMS_Class1XmitTaggedPacket	No	This function reads from Switch Queue 1.
REMS_Class1ReadPacket	No	This function reads from Switch Queue 1.
REMS_DlrEnable	No	This function writes to switch memory.
REMS_DlrDisable	No	This function writes to switch memory.
REMS_EipStartTcu	No	There is only one TCU. Only start the TCU in one place.
REMS_EipEnableBTOIrq	No	This function modifies global data.
REMS_EipDisableBTOIrq	No	This function modifies global data.

Table 152. Modbus/TCP Driver Function Reentrancy

Function	Reentrant	Notes
REMS_Xmit_ModbusTCP_Packet	No	This function writes to Switch Queue 0.
REMS_Read_ModbusTCP_Packet	No	This function reads from Switch Queue 0.

FUNCTION REENTRANCY

Table 153. EtherCAT Driver Function Reentrancy

Function	Reentrant	Notes
HW_Init()	Not applicable	Only called during initialization.
HW_Release()	Not applicable	Only called at end of application.
HW_GetALEventRegister()	No	
HW_GetALEventRegister_Isr()	No	
HW_ResetALEventMask()	No	
HW_SetALEventMask()	No	
HW_DisableSyncManChannel()	No	
HW_EnableSyncManChannel()	No	
HW_GetSyncMan()	No	
HW_EscRead()	No	
HW_EscReadIsr()	No	
HW_EscReadDWord()	No	
HW_EscReadDWordIsr()	No	
HW_EscReadWord()	No	
HW_EscReadWordIsr()	No	
HW_EscReadByte()	No	
HW_EscReadByteIsr()	No	
HW_EscReadMbxMem()	No	
HW_EscWrite()	No	
HW_EscWriteIsr()	No	
HW_EscWriteDWord()	No	
HW_EscWriteDWordIsr()	No	
HW_EscWriteWord()	No	
HW_EscWriteWordIsr()	No	
HW_EscWriteByte()	No	
HW_EscWriteByteIsr()	No	
HW_EscWriteMbxMem()	No	
REMS_ecatMiiEventParams()	Not applicable	Only called in response to interrupt.
REMS_ecatMiiReadComplete()	Not applicable	Only called in response to interrupt.
REMS_ecatMiiWriteComplete	Not applicable	Only called in response to interrupt.
REMS_ecatSetSyncOffsetValue()	Not applicable	Only called during initialization.

ETHERCAT FIDO5200 FUNCTIONAL DIFFERENCES FROM THE BECKHOFF ET1100

The following list addresses the differences in behavior between the fido5200 ESC and the Beckhoff ET1100 ESC. See the data sheet for the ET11000 for more details on register functionality.

- Destroy frame behavior. This difference occurs when a frame must be destroyed in a network. The ET1100 destroys a frame by ending the transmission and adding zero or more bytes until the last four bytes yield an incorrect frame check sequence (FCS), in addition to adding an extra nibble. The fido5200 ends the transmission and adds four bytes, yielding an incorrect FCS, in addition to adding an extra nibble.
- Reset timing. The reset timing for the fido5200 is different from the ET1100 in that the reset is performed immediately by the ET1100. The fido5200 has reset timing that is under the control of the host processor.
- Device description. Descriptions of the registers that interface with the fido5200 and a host processor do not support a 32-bit asynchronous interface when the fido5200 is connected. The description is a 16-bit asynchronous interface.
- Watchdog timing. When the fido5200 is operating in distributed clock mode, the PDI watchdog period is approximated. The difference is less than a 1% error when compared to the ET1100 for the process watchdog.
- FMMU shifted fields. When the ET1100 is used without a host processor, the ET1100 can shift the process data interface (PDU) data arbitrarily when there is a memory access. The fido5200 cannot support arbitrary shifts of the PDU. Instead, the fido5200 shifts a SyncManager status flag to an arbitrary offset within a read PDU. This is the only valid use case for arbitrary PDU shifts.
- DC latch unit not implemented. The fido5200 does not allow the controller to directly read timing for certain actions on the device. This feature is generally limited to certain device types with the implementation determined by the user of the chip. The fido5200 provides latch functionality to the host processor because the fido5200 is typically implemented in more complex devices using the ET1100.
- DC speed counter filter difference register. This register of the ET1100 (Register 0x0932) provides a means to estimate the difference in base oscillator frequency compared to the oscillator frequency on a reference clock (another device on the network). The fido5200 does not generate intermediate values. Instead, the host processor approximates values used to populate Register 0x0932 so that PPM error can be evaluated over the network.
- Enhanced link detection. EtherCAT requires detecting loss of link within a maximum time. Enhanced link detection is a way to allow the usage of a broader range of PHYs. The fido5200 does not implement this feature because PHYs that are used in a design with the fido5200 meet this requirement.
- Reduced signal application time. When the signal is applied to the DLC control word temperature, use Bit 1. When these registers (Register 0x0100 to Register 0x0103) are written, the loop port settings must be applied for approximately a second and then reverted to the previous settings. The apply time of the fido5200 is slightly less than the ET1100.
- Distributed clocks synchronization pulse. There is no specification for the behavior of a synchronization pulse generated by the distributed clocks function. In a four device network, the jitter in Table 154 is observed.
- FMMU configuration. In the fido5200, FMMU registers (Register 0x0600 to Register 0x060F and Register 0x06F0 to Register 0x06FF) cannot be changed while the FMMU is enabled.
- DC speed start register. For the fido5200, the dc speed start register (Register 0x0930) does not impact the dc speed difference register (Register 0x0932).
- AP write (APWR) to 0x0900. APWR does not trigger a timestamp capture on a return path. On the fido5200, any physical write (APWR, FPWR, or BWR) to 0x0900 entering on Port 1 captures a time stamp. The ET1100 only captures a time stamp with an FPWR or a BWR to Register 0x9000 entering Port 1.
- Next Synchronization 0/1 time registers. The ET1100 has shadow registers that contain the content of these registers (Register 0x0990 and Register 0x0998) temporarily if they are read one byte at a time. The fido5200 only reads the data reliably if all eight bytes are read in a single PDU, as opposed to reading one byte at a time.
- Synchronization out register. The synchronization out register (Register 0x0981) for the fido5200 differs from the ET1100 in Bit 3, Bit 5, and Bit 6. Bit 3 only functions as storage. There is no functionality associated with setting this bit. If this bit is set, it reads as set but there is no automatic activation set. Bit 5 and Bit 6 also work as storage, but have some functionality associated with setting these bits. The behavior of Bit 5 and Bit 6 is detailed in Table 155.
- MDIO. The fido5200 does not have pins that correspond to the ET1100 MI_DATA pin or MI_CLK pin. These signals must be connected to the system processor, and the MDIO interface is handled by the processor.
- EEPROM emulation. The fido5200 does not have the signals to communicate directly with an external EEPROM in the same way as the ET1100, and as such, this communication is emulated. See the EEPROM Emulation section for further information.

Table 154. Observed Jitter

Reference Device	ET1100 (ns)	fido5200 (ns)
ET1100 Jitter	20	30
fido5200 Jitter	100	50

ETHERCAT FIDO5200 FUNCTIONAL DIFFERENCES FROM THE BECKHOFF ET1100

Table 155. Observed Behavior of Bit 5 and Bit 6

Bit 5	Bit 6	Observed Behavior
0	0	The data is stored and the device operates the same as the Beckhoff ET1100.
0	1	The data is stored and the device operates the same as the Beckhoff ET1100.
1	0	If the start time is greater than 2^{63} ns, the device starts immediately. The device waits 2^{63} ns until starting instead.
1	1	For start times greater than 2^{31} ns and less than $2^{32} - 1$ ns, the synchronization unit starts immediately. If the start time is greater than $2^{32} - 1$ ns, the synchronization unit is delayed by that amount.

PREBUILD STEPS FOR THE IAR TOOLCHAIN

If a customer is pairing the REM switch with a processor that uses the IAR Toolchain, there are some steps necessary to ensure proper functionality. All of these steps occur as preprocessor build steps. This is due to non-standard compiler behavior on the part of the IAR embedded workbench compiler. By default, the enumeration for REMS_stdIntEvent() is as follows:

```
typedef enum {
  REMS_StdInt_Port_1_LinkChange = REMS_Int_Port_1_Link_Change,
  REMS_StdInt_Port_2_LinkChange = REMS_Int_Port_2_Link_Change,
  REMS_StdInt_PktReady = REMS_Int_Queue_0_Packet_Ready,
  REMS_StdInt_Capture_0 = REMS_Int_Capture_0_Event,
  REMS_StdInt_Capture_1 = REMS_Int_Capture_1_Event,
  REMS_StdInt_Capture_2 = REMS_Int_Capture_2_Event,
  REMS_StdInt_Capture_3 = REMS_Int_Capture_3_Event,
  REMS_StdInt_Compare_0 = REMS_Int_Compare_0_Event,
  REMS_StdInt_Compare_1 = REMS_Int_Compare_1_Event,
  REMS_StdInt_Compare_2 = REMS_Int_Compare_2_Event,
  REMS_StdInt_Compare_3 = REMS_Int_Compare_3_Event,
  REMS_StdInt_TimerControl_0 = REMS_Int_Timer_Control_Event_0,
  REMS_StdInt_TimerControl_1 = REMS_Int_Timer_Control_Event_1,
  REMS_StdInt_TimerControl_2 = REMS_Int_Timer_Control_Event_2,
  REMS_StdInt_TimerControl_3 = REMS_Int_Timer_Control_Event_3,

  /* Protocol-specific interrupt events */
  /* PROFINET */
  REMS_PnetInt_CPM_Watchdog_Timeout = REMS_Int_Host_Port_0,
  REMS_PnetInt_ReceivedRtData = PROTOCOL_INT_ID,
  REMS_PnetInt_Leader_Lost,
  REMS_PnetInt_No_Sync_Message_Received,
  REMS_PnetInt_Jitter_Out_Of_Boundary,
  REMS_PnetInt_Sync,
  REMS_PnetInt_Wrong_Sync_Leader,
  REMS_PnetInt_PortStateRedUp,

  /* EtherCAT */
  REMS_EcatInt_AL_Event_Change = REMS_Int_Host_Port_0,
  REMS_EcatInt_SYNC0_Event  = REMS_Int_Timer_Control_Event_0,
  REMS_EcatInt_SYNC1_Event  = REMS_Int_Timer_Control_Event_1,
  REMS_EcatInt_SYNC0_Int    = REMS_Int_Timer_Control_Int_0,
  REMS_EcatInt_SYNC1_Int    = REMS_Int_Timer_Control_Int_1,
  REMS_EcatInt_SYNCDONE_Int = REMS_Int_Timer_Control_Int_2,
  REMS_EcatInt_MII_MGT_Event = PROTOCOL_INT_ID,
  REMS_EcatInt_Reset_Requested,

  /* TSN */
  REMS_TsnInt_PeerToPeerPktReady = REMS_Int_Queue_0_Packet_Ready,
  REMS_TsnInt_GeneralPktReady = REMS_Int_Queue_1_Packet_Ready,
  REMS_TsnInt_StreamPktReady = REMS_Int_Queue_2_Packet_Ready,
  REMS_TsnInt_EgressTimeAvail_Port1ChanA = REMS_Int_Port_1_0,
  REMS_TsnInt_EgressTimeAvail_Port1ChanB = REMS_Int_Port_1_1,
  REMS_TsnInt_EgressTimeAvail_Port2ChanA = REMS_Int_Port_2_0,
  REMS_TsnInt_EgressTimeAvail_Port2ChanB = REMS_Int_Port_2_1,

}REMS_stdIntEvent_t;
```

PREBUILD STEPS FOR THE IAR TOOLCHAIN

The IAR toolchain does not allow for aliased members of the enumeration. Therefore, the user must take two steps to ensure the enumerations are defined properly when using an ARM processor that uses the IAR toolchain:

1. In the application project settings, define the protocol that is going to be used.
2. When using any event of the enumeration REMS_stdIntEvent_t, look the protocol is defined in the list (per protocol).

Performing these steps allows the user to use the REM switch with an ARM processor using the IAR toolchain.

```
#ifdef REMS_ECAT
/* below is REMS_ECAT version of REMS_stdIntEvent_t */
typedef enum {
  REMS_StdInt_Port_1_LinkChange = REMS_Int_Port_1_Link_Change,
  REMS_StdInt_Port_2_LinkChange = REMS_Int_Port_2_Link_Change,

  /* Protocol-specific interrupt events */
  REMS_EcatInt_AL_Event_Change = REMS_Int_Host_Port_0,
  REMS_EcatInt_SYNC0_Event  = REMS_Int_Timer_Control_Event_3,
  REMS_EcatInt_SYNC1_Event  = REMS_Int_Timer_Control_Event_2,
  REMS_EcatInt_SYNC0_Int    = REMS_Int_Timer_Control_Int_0,
  REMS_EcatInt_SYNC1_Int    = REMS_Int_Timer_Control_Int_1,
  REMS_EcatInt_SYNCDONE_Int = REMS_Int_Timer_Control_Int_2,
  REMS_EcatInt_MII_MGT_Event = PROTOCOL_INT_ID,
  REMS_EcatInt_Reset_Requested
}REMS_stdIntEvent_t;
#endif /* REMS_ECAT *
```

CONSIDERATIONS WHEN USING SIX OR SEVEN FMMUS FOR THE ETHERCAT DRIVER

If the user wants to have six or seven FMMUs programmed by an EtherCAT leader, the user must take additional steps for the FMMUs in their application.

When six or seven FMMUs are programmed, the driver executes the REMS_ECATProgramStoredFMMUEntriesQuick_Finish() function. When REMS_ECAT_ProgramStoredFMMUEntriesQuick_Finish() is called, the time from the rising edge of the high priority interrupt line (Int_2) on the fido5200 to the return of REMS_ECAT_ProgramStoredFMMUEntriesQuick_Finish() must occur in less than 4.30 µs.