RapidIO Switch API™ Application Note

Formal Status

April 13, 2014

About this Document

Topics discussed include the following:

- RapidIO Switch API Introduction
- RapidIO Standard API (DAR)
- IDT Specific API (DSF)
- DAR and DSF Integration
- Example Code

Revision History

April 13, 2014, Formal

- Added Statistics Counter DSF
- Updated Figure 2: DSF File Structure
- Completed other editorial changes as indicated by change bars.

June 6, 2013, Formal

- Added Error Management DSF
- Updated Figure 2: DSF File Structure
- Removed support for CPS Gen1 switches from the API
- Completed other editorial changes

March 22 2011, Preliminary

Added SerDes Configuration API to DSF.

November 3, 2010, Preliminary

First release of the document.

RapidIO Switch API Introduction

The RapidIO Switch API consists of two interfaces:

- The RapidIO Standard API (also known as the Device Access Routines or DAR) – This is an implementation of the functions defined in the *RapidIO Specification (Rev. 2.1) Annex 1: Software/System Bring Up Specification*. These routines use registers defined in the various parts of the *RapidIO Specification (Rev. 2.1)*, and therefore should work for any RapidIO compliant device.

- The IDT Specific API (also known as the Device Specific Functions or DSF) – This presents a common programming model for IDT's Tsi, CPS Gen1, and Gen2 switches. The API includes routines for managing routing tables and port/lane configurations. The DSF is dependent on the DAR for some routines and utilities.

> "Gen2 switches" refers to all IDT CPS and SPS variants.

> API support has been verified for the following devices:
> - CPS-1848
> - CPS-1432
> - SPS-1616
> - Tsi578
>
> Other devices will be added as customers request them.

Both the DAR and the DSF are written in a device and platform independent fashion. Integration of the DAR and DSF requires the implementation of three platform-specific routines:

- A routine to send a maintenance read transaction and return the response data
- A routine to perform a maintenance write transaction
- A routine to delay a requested number of nanoseconds

The DAR and DSF are implemented using the same design pattern. Each consists of a database of routines, some or all of which may be device specific. Each routine in the database is accessed using a handle that identifies the driver for a device, the location of the device, and some constant register values for the device. The platform-specific maintenance read and write routines use the parameters in the handle when accessing the target device.

The parameter style of DAR and DSF routines are distinctly different. DAR routines have multiple parameters of simple types that are compliant with the definitions in the RapidIO specification. In contrast, all DSF routines accept three parameters: the device handle, an input parameter structure, and an output parameter structure. The DSF uses this approach to enable future expansion and modification of the parameters independent of a RapidIO vendor's software deliveries.

All DSF routines return a standard set of return codes. DAR routines that have a return code use the same return code values. All DSF routines also return an implementation-specific return code in the output parameter structure. In the event of a software failure, this return code identifies the location in the DSF code where the failure occurred.
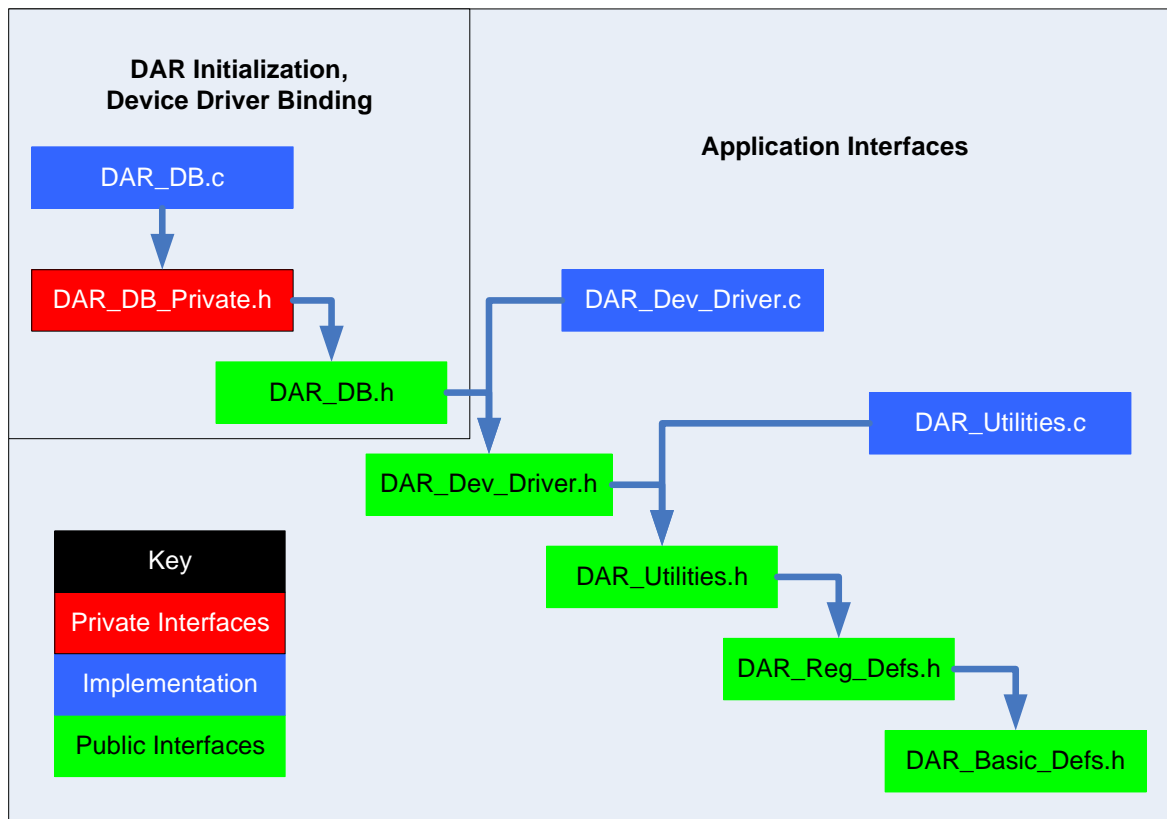
> When customers integrate the RapidIO Switch APIs, it is recommended to incorporate a mechanism for reporting DAR and DSF failure return codes and DSP implementation-specific return codes
> The return codes are used when software support is required.

RapidIO Standard API (DAR)

Figure 1: DAR File Hierarchy

Each file is self documenting internally. The following list contains a brief description and background for each interface file:

- DAR_Basic_Defs.h – This file contains definitions for basic types (such as for unsigned 32-bit integer, UINT32) used by all other DAR and DSF files.

- DAR_Reg_Defs.h – This file contains definitions for the standard RapidIO register offsets and fields used by the DAR and some DSF files. The definitions are coded in a way to ensure they are endian-agnostic.

- DAR_Utilities.h – This file defines utility routines for reading or modifying multiple registers, control symbol composition and parsing, and packet composition and parsing. The control symbol and packet related routines allow translation between a structure that contains parsed field values, and a different structure that contains a stream of bytes. All defined control symbol and packet types and sizes are supported, including CRC computation.

- DAR_Dev_Driver.h – This file is the application interface to the DAR database defined in DAR_DB.h. Applications should always access functions through the interface defined in this file. DAR_Dev_Driver.h supports dynamic creation and destruction of device handles, as is necessary for systems that support hot insertion and removal. DAR_Dev_Driver.h defines all of the return codes used by the DAR and DSF.

- DAR_DB.h – This file is the interface that defines how the DAR database is initialized, and how device driver routines are bound into the DAR database. DAR_DB.h also defines the three platform-specific routines that must be implemented to integrate the DAR onto a specific platform. Drivers can always be added to the DAR database. There are no provisions to remove drivers. The size of the DAR database is controlled by parameters in DAR_DB.h.

  DAR_DB.h contains example code for initialization of the DAR database and binding device drivers.

- DAR_DB_Private.h – This file defines a few macros and details related to device handles and the DAR database that are relevant only to the DAR and DSF.

⚠ Applications are strongly discouraged from using definitions in DAR_DB_Private.h.

- DAR_Utilities.c – This file implements the routines defined in DAR_Utilities.h.

- DAR_Dev_Driver.c – This file implements the routines that will access the device driver routine bound into the DAR DB for a specific function. The routines all have the same design pattern.

- DAR_DB.c – This file implements default routines for every DAR function, based on RapidIO standard registers and RapidIO compliant behavior. "Stub" routines are also implemented, which return a "STUBBED" return code and have no other functionality. After initialization, the last device driver entry in the DAR database contains the default routines. All other device driver entries in the DAR database are initialized with stub routines. The DAR database management routines prevent implementation-specific device drivers from overwriting the last "default" device driver. If no other device driver is found for a device, the "default" device driver is chosen.

The "default" device driver routines are the basis of any new device driver. The driver has the option of overwriting the existing "default" routines with implementation-specific routines.

IDT Specific API (DSF)

The Device Specific Functions, or DSF, contains routines that operate on implementation-specific registers, maximizing performance and functionality. It provides a unified API for the Tsi, CPS Gen1, and Gen2 switch families. The design intent is to keep the API as simple and consistent as possible.

The file structure of the DSF is shown in Figure 2. Note that the DSF currently has the following public interfaces:
- IDT_Port_Config_API.h – This file contains definitions for routines that configure, query, and manage other implementation-specific aspects of port operation.
- IDT_Routing_Table_Config_API.h – This file contains definitions for routines that configure and query routing tables.
- IDT_Error_Management_API.h – This file contains definitions for routines that manage and handle events detected by RapidIO switches.
- IDT_Statistics_Counter_API.h - This file contains definitions to initialize, control, and read statistics counters. Note that application code must decide whether to call the CPS or Tsi control procedures.
- IDT_SerDes_Config_API.h – This file contains definitions for routines that configure, query, and optimize CPS Gen2 SerDes operation (signal quality optimization).
- IDT_DSF_DB_Private.h – This file contains definitions that control which devices are supported by the DSF, as well as the definition of the DSF initialization routine.

There are many private interfaces within the DSF that should not be used by application code (see Figure 2). They are discussed below only to enable users to customize the DSF implementation.

⚠ Applications are strongly discouraged from using any definitions in a private interface since these can change without notice.

Note that each DSF routine has the form of "proc_name(*dev_info, proc_name_in_t *in_parms, proc_name_out_t *out_parms). Dev_info is a device information structure, as defined in DAR_Dev_driver.h. In_parms is a structure containing input parameters for the routine. Out_parms is a structure containing output parameters for the routine.

Note that all out_parms structures contain an "imp_spec" field, which is an implementation specific return code. This return code can be used to identify where and why a fault was detected in a DSF routine.

Figure 2: DSF File Structure
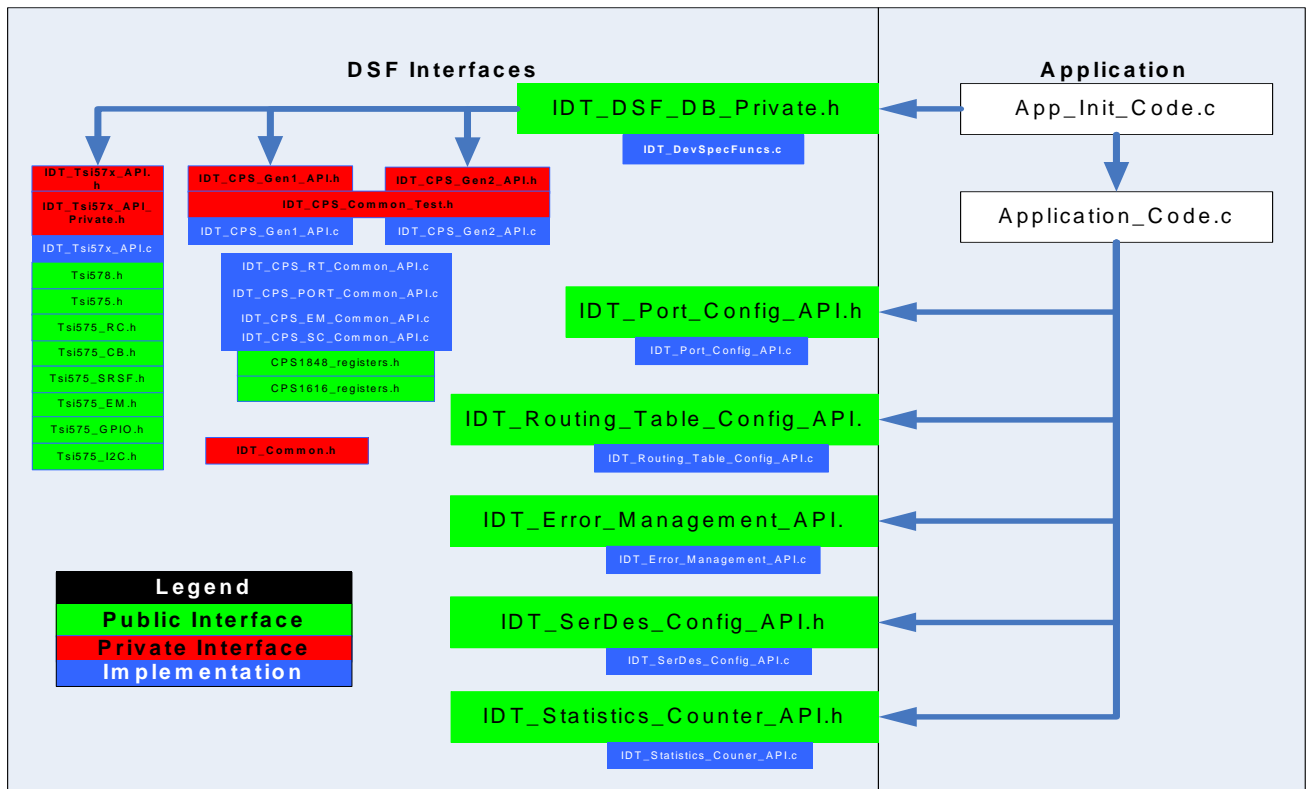
Port Configuration DSF

The IDT Port Configuration (PC) DSF is defined in IDT_Port_Config_API.h, and supports the following functionality:

- Assigns lanes to ports, either one port at a time or for an entire device, along with port width, lane speed, lane swapping, and differential pair inversion
  - idt_pc_set_config - set device configuration
  - idt_pc_get_config - get device configuration
  - idt_pc_get_status - get status of device ports
  - idt_pc_probe - determine if packets can be exchanged on a port
- Resets the link partner and/or the associated port on a device
  - idt_pc_dev_reset_config - control/query how a device/port reacts to an in-band reset request
  - idt_pc_reset_port - resets a port on the local device, optionally attempts to reset the link partnr
  - idt_pc_reset_link_partner - resets the link partner, optionally clears local ackIDs
- Clears errors on a port, including ackID resynchronization
  - idt_pc_clr_errs
- Configures aspects of secure port operation, including acceptance of reset requests from the link partner
  - id_pc_secure_port

The routines support the Tsi and CPSGen2 switches. These devices are designed to connect to a host processor through configuration pin settings or by register settings stored in an I^{2}C EEPROM. The host can then configure the remaining device ports based on an application-specific constant structure. Hot swap events are supported through the ability to reset ports/link partners, and to clear errors.

Lane and port configuration is usually the first step in switch initialization.

The Port Configuration DSF creates a consistent interface for performing these actions on IDT switches; however, the configuration of an individual device is constrained by the capabilities/architecture of that device.
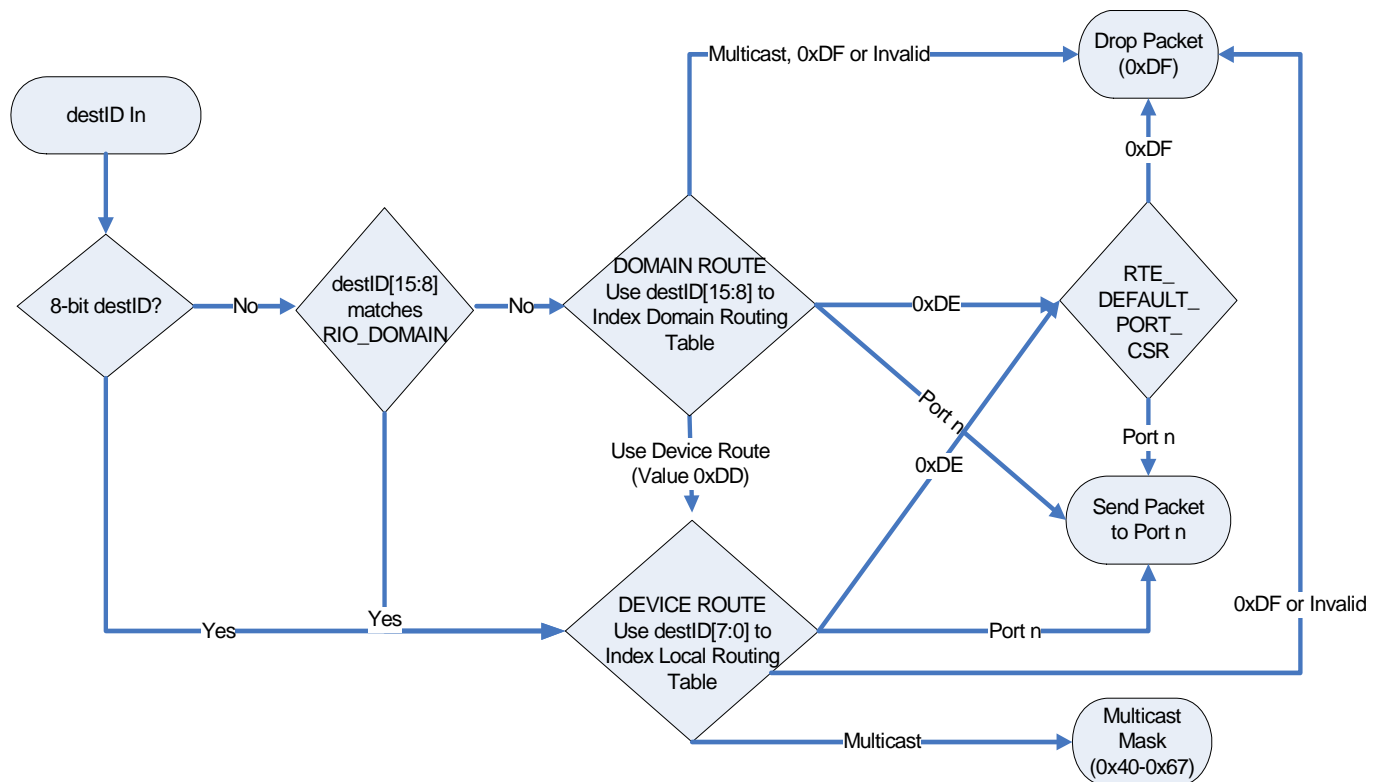
Routing Table DSF

The Routing Table DSF is defined in IDT_Routing_Table_Config_API.h, and supports the following functionality:

- Initializes routing tables
    - idt_rt_initialize - initializes routing table tracking structure and, if requested, updates hardware
    - idt_rt_probe_all - reads current routing table state. NOTE: Restrictions for Tsi57x family.
- Allocates and deallocates multicast resources
    - idt_rt_alloc_mc_mask
    - idt_rt_dealloc_mc_mask
- Sets individual unicast and multicast routes
    - idt_rt_change_rte - change a routing table entry
    - idt_rt_change_mc_mask - change a multicast mask
    - idt_rt_set_all - writes all routing table entries and multicast mask values
    - idt_rt_set_changed - writes routing table entries and multicast mask values changed using idt_rt_change_rte and idt_rt_change_mc_mask.
- Determines the routing of a destination ID (probe)
    - idt_rt_probe - determines handling of packets sent to a destination ID. Can be port specific. If packets are discarded, indicates why the packet is discarded.

The Routing Table DSF supports a programming model that is directly compatible with the CPS Gen1 and Gen2 switch families. As shown in Figure 3, this is an hierarchical model that routes 16-bit destination IDs according to the "Domain" routing table. 8-bit destination IDs and 16-bit destination IDs selected by the "Domain" routing table, are routed according to the "Device" routing table.

Figure 3: Routing Table DSF Programming Model

The Routing Table DSF is consistent for all switch families when a system uses only 8-bit destination IDs. The 16-bit routing table programming model implemented in CPS Gen1 and Gen2 devices is different than that supported by the Tsi switches. As a result, the following restrictions on Routing Table DSF usage apply to 16-bit systems that must support both Tsi and Gen2 switches:

- All routing tables must be initialized after power-up and whenever a port is reset
- Only one Domain routing table entry can select the Device routing table
- Multicast destination IDs must be 8-bit destination IDs, or 16-bit destination IDs with "0" in the most significant byte

⚠ Tsi switches do not directly support the special routing value "0xDF" to drop packets. The DSF supports this by setting the routing table entry with an inverted parity bit, which causes the packets to be dropped.

A side effect of the Routing Table DSF routines is that Tsi switches detect LUT parity errors when packets are dropped.

It is not possible to determine if a Tsi routing table entry is "drop" or "send to default port" by reading it.

The required usage model for these routines is:

1. Initialize all routing table entries on all ports
2. Configure routing for each destination ID in the system

📝 Look-up table configuration is usually the second step in switch initialization.

Port configuration must be completed before routing table configuration can begin.

Error Management DSF

The Error Management (EM) DSF is defined in IDT_Error_Management_API.h, and supports the following functionality:

- Configures parameters for the Port-Write notification mechanism
    - idt_em_cfg_pw - configures port-write reporting. Consistent programming model for all devices.
- Enables/disables and queries which events are detected, and the notification mechanism (interrupt, port-write) for those events
    - idt_em_cfg_set - configures detection of events selected.
    - idt_em_cfg_get - queries configuration state of events selected
    - idt_em_dev_rpt_ctl - configures when mechanism is used for reporting events
- Parses received port-write data
    - idt_em_parse_pw - parses both standard port writes as well as IDT CPS Gen2 specific port-write format
- Determines which events have been detected and clears the detected events
    - idt_em_get_int_stat - finds events detected and reported through the device interrupt pin.
    - idt_em_get_pw_stat - finds events detected and reported through port-write packets
    - idt_em_clr_events - clears evens found by idt_em_get_int_stat and idt_em_get_pw_stat
- Creates events for software test purposes
    - idt_em_create_events - allows all events to be created for test purposes.

The Error Management DSF defines events that are supported by Tsi and CPS Gen2 devices. CPS Gen1 devices are not supported because they have few error management features. The events are categorized in three severity levels:

- Fatal – It is not possible to exchange packets on the link. When detection of these events is enabled, the Error Management DSF configures the device to automatically discard packets to prevent a cascade congestion failure of the system. For example, the removal of a link partner is a fatal error in most systems.
- Drop – A packet has been dropped due to an error. An example "drop" event occurs when a switch receives a Port-Write that has a hop count value of zero.
- Informational – An event has occurred that the system may want to know about. For example, an informational event can be detected when a link initializes.

The Error Management DSF provides a toolbox for many different error management system designs. One example design is summarized as follows:

- Initialize system event management by configuring the Port-Write notification mechanism if required, enabling events that the system must handle, and setting the notification mechanism
- When an event notification (interrupt or port-write) is received, the system dispatches the event to a software entity for handling. This is reasonable because event handling is not real-time critical.
- The software entity uses the Error Management DSF to parse the port-write, to determine the list of outstanding events, and to clear those events.
- The software entity may also use the Port Configuration DSF and Routing Table Configuration DSF to adjust system operation based on the events received.

Signal Quality DSF

The Signal Quality API is defined in IDT_SerDes_Config_API.h. It consists of routines that perform the following functions:

- SerDes Transmit Parameter range query, value query, and set
    - idt_sq_get_tx_coeff_range
    - idt_sq_get_tx_config
    - idt_sq_set_tx_config
- SerDes Receive Parameter (DFE) range query, value query, and set
    - idt_sq_get_rx_coeff_range
    - idt_sq_get_rx_config
    - idt_sq_set_rx_config
- Configure and read bit error rate counters for a port
    - idt_sq_config_ber_ctr
    - idt_sq_get_ber_ctr
- Scan SerDes receive parameter values to determine optimal values
    - idt_sq_ber_scan_status_init
    - idt_sq_ber_scan_update
- Monitor signal quality (received bit error rate) for a lane
    - idt_sq_ber_mon_status_init
    - idt_sq_ber_mon_update
- Adjust SerDes Receive Parameter values if the monitored signal quality does not meet required standards
    - idt_sq_ber_adj_status_init
    - idt_sq_ber_adj_update

The SerDes Configuration API is intended for use only with links whose signal characteristics are between medium and long reach according to the *RapidIO Specification (Rev. 2.1)*, Part 6. Only IDT's Gen 2 switches are supported by the SerDes Configuration API. For more information on signal quality optimization, see the *IDT Signal Quality Optimization Application Note*.

The SerDes Configuration API is intended to be used as follows:

- SerDes Transmit and Receive parameters must be set before a port is used. These parameters can be set from an EEPROM connected to the device, or by system host software during system initialization. Bit error rate counter configuration is also performed as part of system initialization.
- The SerDes Transmit and Receive parameters must be determined before system initialization. The RapidFET-JTAG tool, which is available at www.fetcorp.com, can be used to determine SerDes transmit and receive parameters. SerDes receive parameters can also be determined before system initialization through execution of the SerDes Configuration API Scan function.
- Variations in operating and manufacturing may result in the need to adjust the initial SerDes receive parameters. SerDes receive parameters can be adjusted by repeatedly calling the idt_sc_ber_adj_update() routine with the bit error rate counter value.
- During normal system operation, the signal quality of each lane can be checked periodically by reading the bit error counter and calling the idt_sc_ber_mon_update() routine. This process is called "monitoring". The idt_sc_ber_mon_update() routine implements a leaky bucket for bit errors. If the programmed maximum bit error rate is exceeded, the idt_sc_ber_mon_update() routine may optionally invoke SerDes receive parameter adjustment.

Statistics Counter DSF

The Statistics Counter API is defined in IDT_Statistics_Counter_API.h. It consists of routines that perform the following functions:

- Initialize the statistics counters tracking structure
    - idt_sc_init_dev_ctrs
- Read the statistics counters
    - idt_sc_read_ctrs
- Configure Tsi57x programmable statistics counter
    - idt_sc_cfg_tsi57x_ctr
- Enable/disable CPS Gen2 statistics counters
    - idt_sc_cfg_cps_ctrs

Each statistics counter is modelled as a 64 bit value which captures the counter value since it was last initialized/configured, and a 32-bit value that reflects the amount the counter has increased since last read.

The values counted are described through three values:

- idt_sc_ctr_t - an enumeration of the types of things that can be counted
- BOOL tx:whether the thing counted is being transmitted (TRUE) or received (FALSE) by the device)
- BOOL rio: whether the thing being counted is on the RapidIO interface (TRUE), or an internal interface (FALSE) of the device

For example, one possible configuration of a Tsi57x counter is { idt_sc_uc_pkts, TX, RIO}. This counts all unicast packets transmitted by the Tsi57x. An example of a CPS Gen2 counter is {idt_sc_pkt, RX, !RIO}. This counts all packets received from the RapidIO switch fabric by a port.

DSF Implementation

The DSF is implemented using a design pattern similar to the DAR. All driver routines are bound into a database. The database is accessed through routines defined in the interfaces (IDT_Port_Config_API.h, IDT_Routing_Table_Config_API.h, IDT_Error_Management_API.h, IDT_Statistics_Counter_API.h and IDT_Serdes_Config_API.h).

All IDT routines are bound into the DAR and DSF by routines defined in IDT_DSF_DB_Private.h. IDT_DSF_DB_Private.h is named as a private interface to prevent application code from using it. This interface should be used only by application initialization code. The DSF DB is implemented in IDT_DevSpecFunctions.c.

The remaining code implements the DSF. The code is provided to enable debugging within customers systems. Application code may make use of Tsi, CPS Gen1, and Gen2 register definitions.

⚠ Application code is cautioned against using any utility routines and definitions that are internal to the DSF.

IDT may change the internal implementation and definitions of the DSF without notice. The public interfaces will remain constant, and forward/backward compatible.

The remaining files are categorized as follows.

CPS Gen1/Gen2 Register Definitions

CPS1848_registers.h, CPS1616_registers.h.

CPS1848_registers.h definitions are used for functions common between the CPS-1848 and CPS-1616. CPS-1616 definitions are used for functions specific to the CPS-1616.

Tsi Register Definitions

tsi578.h, Tsi575.h, Tsi575_CG.h, Tsi575_EM.h, Tsi575_GPIO.h, Tsi575_I2C.h, Tsi575_RC.h, Tsi575_SRSF.h

Tsi575 definition files are for the switch component of the Tsi620.

Internal Private Interfaces

IDT_Common.h, IDT_CPS_Common_Test.h, IDT_Tsi57x_API_Private.h, IDT_CPS_Gen1_API.h, IDT_CPS_Gen2_API.h

CPS Gen1/Gen2 DSF Implementation

IDT_CPS_PORT_Common_API.c, IDT_CPS_RT_Common_API.c, IDT_CPS_EM_Common_API.c, IDT_CPS_Gen1_API.c, IDT_CPS_Gen2_API.c, IDT_SerDes_Config_API.c

Tsi DSF Implementation

IDT_Tsi57x_API.h, IDT_Tsi57x_API.c

DSF Implementation

IDT_DevSpecFuncs.c, IDT_Port_Config_API.c, IDT_Routing_Table_Config_API.c, IDT_SerDes_Config_API.c

DAR and DSF Integration

The DAR and DSF are both initialized by calling IDT_DSF_bind_DAR_routines(). IDT_DSF_bind_DAR_routines accepts three functions, a ReadReg routine, a WriteReg routine, and a WaitSec routine.

The functionality required of these routines is well documented within the DAR_DB.h and DAR_Utilties.h interfaces.

ReadReg and WriteReg must not be called directly by application code. Application code should always call DARRegRead and DARRegWrite.

The WaitSec routine can be called directly by application code.
The WaitSec routine needs to support a granularity of microseconds.
The WaitSec routine must wait at least as long as the period requested.

Mutual Exclusion

The DAR and DSF routines do not incorporate mutual exclusion support. Application code is responsible for implementing mutual exclusion support for two aspects of system operation:

1. A device should only be manipulated from one processor at a time. This implies the standard routines for register access mutual exclusion, DARrioAcquireDeviceLock and DARrioReleaseDeviceLock, must be used by the application code.

2. Some sequences of register accesses, such as reading or writing routing tables, must occur sequentially without interruption. Examples of these are routing table manipulation routines for both the DAR and DSF. For this reason, calls to DAR and DSF routines must ensure that they are allowed to run to completion without interruption.

Interrupts and Exceptions

Register reads and writes may time out as part of normal system discovery and error handling operation. For many systems, these timeouts occur in about 10 microseconds. Users may find that it is most efficient to perform an in-line "busy wait" rather than to handle an interrupt for such short timeout intervals.

If maintenance packets are generated by a processor read, exceptions that occur due to response timeouts must be handled. This implies integrating interrupt support into the DAR ReadReg and WriteReg routines.

> The DAR_DEV_INFO_t structure has two pointers, privateData and accessInfo, which can be used by the application/platform to integrate support for register access related interrupts and exceptions into the DAR and DSF.

The DAR and DSF routines are reentrant. They can, therefore, be called from within interrupt handlers or by application code.

> The number of register accesses performed by a DSF routine, particularly for routing table manipulations, may be large. This is particularly true of routines that read or write the entire routing table of the device. This can impact interrupt performance.

Example Code

The APIs are delivered with an "example_code.c" file, which contains examples of the following:

- initialization of the RapidIO Switch API, including example types, register read/write routines, and delay routine.
- initialization of a RapidIO switch, including port configuration, routing table initialization, and event management configuration
- Statistics Counter API usage

April 13, 2014