Table of Contents

Introduction

Linux supports the z/Architecture Channel Subsystem feature Queued Direct Input Output (QDIO). This document describes the hardware capabilities as derived from the Linux implementation of QDIO supporting devices.

QDIO requires cooperation between the program, Linux, the Channel Subsystem, and QDIO supporting device. Linux supports two devices that utilize QDIO:

Open System Adapters (Linux qeth devices) and

Fiber Channel Protocol SCSI disks (Linux zfcp devices).

The facilities provided by QDIO are fundamentally the same, although the device implementations vary. This document focuses primarily on QDIO from the perspective of its use by Linux qeth devices, but both zfcp (to a lesser degree) and qeth (to a much greater degree) usages were explored to understand QDIO.

Group Device

A “group device” presents to the program multiple subchannels that operate in concert to provide a single set of functionality.

In the case of QETH group devices, their are two devices, the read and write subchannels, devoted to network related configuration information exchanges. If the read subchannel has a device address of n, then the write subchannel must have a device address of n+1. The

third subchannel in the group, the QDIO data device may be configured with any device address. The QDIO data device address is independent of the read and write subchannel device addresses.

Adapter

This document uses the term “adapter” to refer to a single instance of a device presenting to the CPU one or more subchannels of which one (or the only) subchannel is a QDIO data device.

In the case of an OSA (Open System Adapter) Express hardware device, each network port constitutes a single adapter. By the above definition of “adapter,” an OSA Express or OSA Express2 device supports at most two adapters and an OSA Express3 device supports at most four adapters.

Features and Facilities

Queued Direct Input/Output Feature (QDIO)

QDIO channel subsystem feature provides:

A new device, the data device,

A new CPU instruction, SIGNAL ADAPTER, that allows the program to notify the adapter of program events and synchronize state with the adapter,

Various structures understood by the QDIO feature,

A channel subsystem call that queries a data device on behalf of the program for QDIO information and

Reporting of channel subsystem QDIO capabilities to the program.

Design Limits

The QDIO implementation appears to have the following limits or values:

Maximum input queues per data device – 32

Maximum output queues per data device – 32

Number of buffers per queue – 128

Maximum number of entries per SBAL – 16

Maximum non-shared adapter interrupt indicators – 63

Shared adapter interrupt indicators - 1

QDIO Data Device

The data device provides the linkage between the program, the channel subsystem QDIO feature and a hardware adapter that supports QDIO functionality. By reporting SENSE ID data consistent with the adapter with which a specific data device is associated, the data device appears to be a functioning component of the adapter. The data device is in reality implemented by the channel subsystem.

Two CCW commands are provided by the data device: ESTABLISH and ACTIVATE. The ESTABLISH command provides the communication linkage between the program and the data device, defining for the channel subsystem how the program will use the facility. Once the program interface with the QDIO feature is established, the interface between the channel subsystem and the hardware adapter may be started by the program issuing the ACTIVATE command to the data device. Following activation of the adapter, the program may utilize the new facilities (CPU instruction interrupts and structures) provided by the QDIO feature for

communication with the adapter.

The fundamental unit of communication between the program and the adapter is a buffer associated with a queue. Conceptually the queue is a circular ring of 128 buffers. Buffers are shared between the adapter and program as each proceed around the ring. A buffer is defined by its associated System Buffer Address List. This list is composed of one or more entries that relate to the buffer. The QDIO feature provides access to the buffer's address list by the adapter. The adapter's own path to main storage allows the address list to be read or written by the adapter as can the buffer contents via the addresses supplied by the program in the list itself.

Adapter Interrupt Facility (AIF)

AIF provides a:

new form of input/output interrupt independent of a device that informs the program of an adapter event,

storage resident indicator that allows the program to recognize the occurrence of an adapter event on

any adapter by use of a summary indicator or

an adapter associated with a specific QDIO data subchannel's adapter.

new channel subsystem request allowing the indicator for a specific data device to be set and/or a summary indicator to be identified, and

new CPU privileged instruction, SVS, giving the program the ability to manage channel subsystem resident indicators. The instruction name associated with the SVS mnemonic is not used. Herein, it will stand for the SUBSYSTEM VARY STATUS instruction.

QDIO Extended State Block Management Facility (QEBSM)

QEBSM provides two new privileged CPU instructions designed to improve performance when the program is managing buffer state information in a z/VM environment:

SET QUEUE BUFFER STATE (SQBS) and

EXTRACT QUEUE BUFFER STATE (EQBS).

A token associated with the subchannel provided to the program via a CHANNEL SUBSYSTEM CALL request.

Use of the subchannel token when identifying the data device to the SIGNAL ADAPTER instruction.

Use of QEBSM eliminates the need for SIGNAL ADAPTER Synchronize commands intercepted by z/VM. An SIE feature, QIOAssist, provides direct support of SIGNAL ADAPTER Input or Output requests and handling of QEBSM.

Reference

SC24-6081-05, z/VM V5R3.0 CP Commands and Utilities Reference, commands QUERY QIOASSIST, QUERY VIRTUAL OSA

z/VM V5R2 Performance Report, http://www.vm.ibm.com/perf/reports/zvm/html/qebsm.html

Time Delay Disablement Facility (TDD)

TDD extends AIF with the ability to disable the built-in time delay used to defer PCI input interrupts until input slows or input buffers are exhausted. Disablement of the input interrupt time delay ignores the BLKT value specified in the adapter parameter field during queue establishment.

QDIO Operations Overview

QDIO links a program and an adapter for input/output activities. As such, QDIO has two interfaces:

A program to channel subsystem interface and

a channel subsystem to adapter interface.

Only the first is visible to the program. The second is completely embedded in the System z hardware. While the second may influence how the program operates via the channel subsystem adapter interface, it is impossible to differentiate whether the channel subsystem interface the program observes is an artifact of QDIO operations or how the adapter interfaces with the channel subsystem. This phenomenon is observed in system buffer access list usage by the adapter.

Fiber channel buffers always have sixteen access list entries per buffer and network adapter buffers may have as few as one access list entry per buffer. Network adapter buffer access lists appear to be read-only from the perspective of the adapter. In other words, network adapters do not alter the content of an access list entry supplied by the program, while the fiber channel adapter does.

There is only one place where the channel subsystem is explicitly exposed to the nature of the adapter during QDIO initialization. When the queues are described to the channel subsystem by the program via the data device (notice all three participants are linked in this operation), a parameter, queue format, is provided by the program that explicitly identifies whether the queues are network adapter queues or fiber channel queues. Exactly how this parameter effects QDIO operations is unclear, but it definitely provides the opportunity for choice by the program and specification of adapter specific information.

This suggests that these variances in QDIO operations between the two different adapters is the result of implementation choices made by the adapter developers in their internal use of the adapter to channel subsystem interface.

Before a QDIO data device may be utilized it must be initialized with the program's configuration. Once initialized and activated input/output operations may proceed until the device is halted. Output operations are always initiated by the program. Input operations may be under program control (zfcp) or occur unsolicited (qeth) depending upon the adapter.

Recognition of the completion of input or output operations is provided by a CPU recognized I/O interruption. The source of the interruption may be either the data device subchannel or directly by the adapter. Output operation completion is only provided by a program controlled interrupt (PCI) provided to the program via the data device when requested by the program. The program requests this by an indication in a SBAL entry. Completion of input operations may be provided be either the adapter directly or data device depending upon how the data device has been configured for operation.

Following adapter initialization and activation, input operations may then be solicited by the program. When input operations are not expressly solicited by the program, recognition by the program that input has occurred is provided by one of two input/output interrupt types. Output operations are always initiated by the program and completion of the output is provided via an input interruption triggered by the data device if requested by the program and supported by the adapter.

Adapter Initialization

Initialization of the adapter falls into three distinct phases:

Recognizing the adapter group device

Configuring the adapter for operations and

Configuring QDIO operations.

Adapter Recognition

The program must recognize any group devices and those that support QDIO by means of the extended SENSE ID channel command issued to each subchannel. The extended SENSE ID will provide the channel commands required to complete the data device initialization via CIW's.

The SENSE ID command is used to determine if the group device is an:

OSA Express Adapter or

Hipersocket Adapter or

OSA for NCP adapter.

Each device in the group will respond with the same basic identification information. The QDIO data device is further probed using the READ CONFIGURATION COMMAND to determine if the adapter is a physical OSA Express or z/VM Guest LAN.

Network Adapter Operations

Adapter operations configure the adapter for its intended usage. For a network adapter, this would include parameters related to either Layer-2 or Layer-3 operations. Adapter operation initialization occurs over what will be called here, the adapter interface.

The adapter interface utilizes the adapter's read/write subchannels as a single bi-directional communication path. The adapter interface communication path is established by the program issuing to each of the read and write subchannels a:

1. WRITE CCW command containing an ID EXCHANGE activation command as data followed by a
2. READ CCW command containing as data the subchannel's response to the ID EXCHANGE adapter command.

Among other things, this process validates that the subchannels the program believes to be the adapter's read and write subchannels are in fact the read and write subchannels for the adapter.

Once this adapter interface has been established, configuration of the adapter's operations can be performed by the program issuing various commands to the adapter that configure its operation. Details of these exchanges are not documented here as they are independent of the QDIO operations discussed in this document. A companion document provides these details.

QDIO Operations

The program must query the channel subsystem to determine:

available QDIO features and

the specific QDIO capabilities of individual attached adapter data devices.

Before data devices may be used, some group devices will require other operational aspects of the adapter to be initialized. Such initialization must occur before an adapter is initialized for QDIO input and output operations. For network adapters, operational aspects include various configuration related to either Layer-2 or Layer-3 operations. Adapter configuration is performed using the adapter's write and read devices operating as a single interface channel.

Queues, as managed by the program, are defined to the data device by means of the ESTABLISH channel command. Input/output operations are started by the ACTIVATE channel command. Exactly what actions are performed by the ACTIVATE channel command is not clear from the Linux code. However the following can be inferred:

I/O operations will not occur until the data device has been activated and

an activate check condition can occur which suggests some form of error detection occurs during activation.

Following queue initialization and activation, the data device may be used by the program for QDIO input and output operations with the SIGNAL ADAPTER instruction and respond to Program Control and Adapter Interrupts, and QEBSM instructions if available.

Queue Establishment

The program must create all of the structures required by the data device. For Linux this process is separated between the adapter driver and the QDIO driver.

The Linux adapter driver creates:

arrays (one per queue) of SBAL pointers used by the SL array.

QIB adapter parameter field content and

an initialization structure that includes:

the queue format,

number of input and output queues,

an interrupt parameter identifying the address of the QETH structure associated with the adapter

queue handlers

and operational flags, for example, use output PCI's, and others

The QDIO driver establishes the following structures:

SLIB,

SBAL (SBAL entries are not complete, see “Input Preparation” and “Output Preparation” sections, below),

Input buffers are set to SLSB_P_INPUT_NOT_INIT,

Output buffers are set to SLSB_P_OUTPUT_NOT_INIT,

QIB,

QDR,

Adapter interrupt indicators, if used, and

issues the ESTABLISH and ACTIVATE commands to the data device.

Reference example:

drivers/s390/net/qeth_core_main.c/qeth_qdio_establish
drivers/s390/cio/qdio_main.c/qdio_establish
drivers/s390/cio/qdio_setup.c/setup_irq (and various called functions)
drivers/390cio/qdio_thinint.c/qdio_establsh_thinint

Input Preparation

The program must prepare the buffers that will receive input before QDIO can provide data in them. See the “Input Buffer Usage” section, below.

The prepared buffers must then be given to the data device for use by setting the SLSB to a state of SLSB_CU_INPUT_EMPTY. If the data device has no available input buffers the program must issue a SIGNAL ADAPTER Input function to make the adapter aware of the availability of input buffers if the adapter characteristic values indicate this requirements.

Input Buffer Usage

The Linux network adapter support initializes the SBAL entries with the following values:

SBAL Entry Field	Entries	Value	Notes
length	0-15	4096	
address	0-15	Pool entry element address	
flags	0-14	0	

SBAL Entry Field	Entries	Value	Notes
	15	0x40	Flag this is last entry

Reference example:

drivers/s390/net/qeth_core_main.c/qeth_init_qdio_queues
drivers/s390/cio/qdio_main.c/handle_inbound

Input Recognition

The program is signaled by the adapter that input is available by means of either a Programmed Control Interrupt (the default) or an Adapter Interrupt (if available). The purpose of both is to inform the program of presence of inbound buffers that require processing.

Programmed Controlled Interrupt Signaling

Whenever a PCI interrupt is received from a data device, each established input queue, usually only one, must be examined to determine if input has been received. The SLSB is examined to determine this based upon the buffers previously provided to the adapter for input. The SLSB does not require the SIGNAL ADAPTER Synchronize function because synchronization is automatically performed by the adapter for inbound frames or packets.

The buffers states examined for buffers used for output operations are:

SLSB_P_INPUT_PRIMED (0x82) – Input has been received

SLSB_P_INPUT_ERROR (0x8F) – Error has occurred processing this inbound buffer

SLSB_CU_INPUT_EMPTY (0x41) – Buffer has been provided for input, still empty

SLSB_P_INPUT_NOT_INIT (0x80) – Buffer has not been prepared for input

SLSB_P_INPUT_ACK (0x81) – Program acknowledges buffer contains inbound data and processing is pending completion

Reference example:

drivers/s390/cio/qdio_main.c/qdio_int_handler
drivers/s390/cio/qdio_main.c/qdio_int_handler_pci
drivers/s390/cio/qdio_main.c/qdio_inbound_processing

Adapter Interrupt Signaling

Adapter interrupts on not device specific. It it the responsibility of the program to validate each input queue to identify which has received input. Examination of the subchannel or summary indicators may be used, or some other program specific mechanism. Once an indicator has been identified as requiring service, it must be cleared. Clearing requires an atomic operation. If the shared indicator has identified one or more subchannel requiring servicing of its input queue, following recognition for each such subchannel, the shared

indicator must be cleared using the SUBSYSTEM VARY STATUS instruction.

Reference example:

drivers/s390/cio/airq.c/do_adapter_IO
drivers/s390/cio/qdio_thinint.c/tiqdio_thinint_handler
drivers/s390/cio/qdio_main.c/tiqdio_inbound_processing

Input Processing

Once a buffer is recognized as containing input to be processed, the buffer is processed by the QETH Layer-2 or Layer-3 inbound queue handler. Following the processing of the inbound buffer, the SLSB state is set to SLSB_CU_INPUT_EMPTY. If the adapter has run out of free input buffers, SIGNAL ADAPTER Input should be issued. SIGNAL ADAPTER Synchronize function is then performed to both update the adapter with processed buffers and recognize new input.

Reference example:

drivers/s390/net/qeth_l2_main.c/qeth_l2_qdio_input_handler
drivers/s390/net/qeth_core_main.c/qeth_queue_input_buffer
drivers/s390/cio/qdio_main.c/do_QDIO
drivers/s390/cio/qdio_main.c/handle_inbound

Output Preparation

QETH buffer preparation is dependent upon two factors:

TCP Segmentation Offload is in use or

Packing mode is active.

When TCP Segmentation Offload is in use, the first SBAL entry will contain data pointing to the packet header and reflect its length.

When packing mode is in use, multiple complete packets or frames will be contained in a single buffer. However, each packet or frame will be pointed to by one of more SBAL entries, meaning that an individual packet or frame will not share an SBAL entry with another packet or frame.

The last used entry of the SBAL will have the last entry flag set in bit 1. If the completion of processing by the adapter of this SBAL is to generate a PCI interrupt, the PCI flag in bit 25 of at least one of the entries must be set. This may be the first or last.

Once the SBAL entries have been completely defined for the buffer, its SLSB is set to SLSB_CU_OUTPUT_PRIMED (0x62).

Reference example:

drivers/s390/net/qeth_core_main.c/qeth_do_send_packet
drivers/s390/net/qeth_core_main.c/qeth_fill_buffer
drivers/s390/net/qeth_core_main.c/__qeth_fill_buffer

drivers/s390/cio/qdio_main.c/do_IRQ
drivers/s390/cio/qdio_main.c/handle_outbound

TCP Segmentation Offload Buffer Usage

The first SBAL entry isolates for the adapter the packet header.

SBAL Entry Field	Entries	Value
length	0	Packet header size
	1-15	4096 or TCP data fragment length
address	0	Packet header address
	1-15	TCP data fragment address start
flags	0	First fragment
	1-15	Middle or last fragment
	last	Bit 1 set to 1, indicating last entry
	15	Bits 24-31 not zero indicating a send error (only used by Hipersocket)

Non-packing Mode Output Buffer Usage

As many entries as needed are used to point to the fragments of the buffer as necessary. Only one packet or frame is identified in the buffer.

SBAL Entry Field	Entries	Value
length	0-15	4096 or fragment length
address	0-15	Fragment start address
flags	0-15	First, middle or last fragment
	last	Bit 1 set to 1, indicating last entry
	15	Bits 24-31 not zero indicating a send error (only used by Hipersocket)

Packing Mode Output Buffer Usage

Multiple complete QETH packets or frames may be sent in a single buffer by using as many SBAL entries as required and available to do so. A separate SBAL entry is required to define the starting (or only) fragment of each packet or frame being sent. This means that a SBAL entry will provide location and size information for one and only one packet or frame at a time.

The following is an example of the SBAL entry content when packing mode is in use:

Entry	Last Entry Flag	Fragment Flag	PCI Flag
0	0	Packet 1 - First	1
1	0	Packet 1 - Middle	0
2	0	Packet 1 - Last	0
3	0	Packet 2 - Last	0
4	0	Packet 3 - First	0
5	0	Packet 3 - Last	0
6	1	Packet 4 - Last	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15 *	0	0	0

* Hipersocket may post an error indication in bits 24-31 of the flag field in this entry. This will be accompanied by an error state being indicated in the SLSB entry associated with the error SBAL.

Output Initiation

To provide output to the adapter on an output queue, the program must prepare the buffer(s) for output (see “Output Preparation” section above), treating the output buffers as a ring.

If the adapter characteristics indicates that a SIGNAL ADAPTER Synchronize function is needed, the function is used to update the adapter with the buffer states of the primed output buffers. In the case where PCI interrupts are not supported by the adapter, this Synchronize function will provide the program opportunity to recognize the completion of output for previously primed output buffers. This is recognized by the SLSB state changing from SLSB_CU_OUTPUT_PRIMED to SLSB_P_OUTPUT_EMPTY.

If the previously initiated output operation is still in progress (indicated by the buffer previous to the first being sent for output is still control unit primed), no further action is required of the program. The newly added buffers will be output when reached by the adapter.

If the previously initiated output operation has completed (indicated by the previous buffer

preceding the first being sent for output is no longer primed), the program will issue a SIGNAL ADAPTER Output function to initiate the output operation.

Reference example:

cio/qdio_main.c/handle_outbound

Output Completion

When an output buffer SBAL flag indicates that a PCI is to be generated, a PCI I/O interrupt will be generated by the data device indicating to the program that a QDIO event has occurred. The PCI interrupt will occur at completion of the processing of the SBAL. Upon recognizing the program controlled interrupt from the data device, the program must determine which of the devices output queues have completed. If automatic synchronization on an output triggered PCI is not provided by a hypervisor, then the program must issue the SIGNAL ADAPTER Synchronize function with the output queue mask of the output queue for which output completion is expected.

The buffers states examined for buffers used for output operations are:

SLSB_P_OUTPUT_EMPTY (0xA1) – successfully output by the adapter

SLSB_P_OUTPUT_ERROR (0xAF) – unsuccessfully output by the adapter

SLSB_CU_OUTPUT_PRIMED (0x62) – still waiting to be output by the adapter

SLSB_P_OUTPUT_NOT_INIT (0xA0) – associated SBAL not initialized

SLSB_P_OUTPUT_HALTED (0xAE) – not output because the data device was halted.

It is also possible for a PCI interrupt to be missed(?) due to adapter interrupt support. During adapter interrupt handling, output queues should be synchronized (if needed) and checked for output completion. This suggests that the triggering of an adapter interrupt has the effect of clearing a pending PCI interruption as well.

Reference example:

drivers/s390/cio/qdio_main.c/qdio_int_handler_pci
drivers/s390/cio/qdio_main.c/__qdio_outbound_processing
drivers/s390/cio/qdio_main.c/get_outbound_buffer_frontier
drivers/s390/cio/qdio_main.c/qdio_outbound_q_moved
drivers/s390/cio/qdio_thinint.c/tiqdio_thinint_handler

Networking Facility and Feature Usage

Fundamentally three networking environments utilize the QDIO feature and its related facilities:

1. Two networking environments are available to a guest running in a PR/SM LPAR:

   Any of three generations of Open System Adapter Express hardware components (OSA Express, OSA Express 2 or OSA Express 3) or

   Hipersockets provided by PR/SM, a form of OSA emulation.

2. One networking environment is provided to a guest running in z/VM

   Guest LAN, another variant of OSA Express emulation, with or without VSWITCH.

The key differentiators of these environments from a QDIO perspective are the feature and facility usage. These are articulated primarily by means of the adapter characteristics field, qdioac1 (byte 6, of the Subsystem Queue Description data) provided by the data device subchannel.

Bit	qdioac1	OSA	OSA 2	OSA 3	Hipersocket	Guest LAN
AC 1	SIGA_INPUT_NEEDED	1	1	1	1	0
AC 2	SIGA_OUPUT_NEEDED	1	1	1	1	0
AC 3	SIGA_SYNC_NEEDED	1	1	1	0	0
AC 4	AUTO_SYNC_ON_THININT	0	0	0	1	1
AC 5	AUTO_SYNC_ON_OUT_PCI	0	0	0	1	1
AC 6	QEBSM_AVAILABLE	0	0	0	0	1
AC 7	QEBSM_ENABLED	0	0	0	0	1
GC 41	QDIO	1	1	1	1	1
GC 56	AIF_TDD	0	0	1	0	0
GC 58	QEBSM Available	0	1	1	1	1
GC 67	AIF_OSA	0	1	1	1	0

Assigned Storage

I/O Interruption Identification Word

When an adapter interrupt has occurred, the I/O interruption identification word indicates this by setting bit 0 to 1. The adapter interrupt type is identified in bits 17-19.

Normal I/O interrupts have bits 0,1 and 5-31 set to zeros.

The I/O interruption identification word is stored at real address 192 or x'C0'.

Bit Usage (0-15)	Linux Symbol	Description
0...	adapter_IO	Subchannel I/O interrupt
1...	adapter_IO	Adapter interrupt
.X..		reserved
..XX X...	isc	I/O interrupt subclass
.... .XXX XXXX XXXX		Reserved

Bit Usage (16-31)	Linux Symbol	Description
X...		reserved
.XXX	int_type	Adapter interrupt type, 0==I/O
.... XXXX XXXX XXXX		Reserved, bits 20-31

Channel Subsystem Calls

Channel Subsystem Characteristics Call

Request/Response Layout

Disp.	Length	Field	Description
+0	2	req.length	Request length (0x0010, 16 bytes)
+2	2	code	Request code (0x0010)
+4	12		reserved
+16	2	rsp.length	Response length
+18	2	rsp.code	Response code
+20	4		reserved
+24	2040	general_char	General characteristics
2064	2072	chsc_char	Channel subsystem characteristics

General Characteristics

The general characteristics constitutes a series of bits indicating the general characteristics of the channel subsystem. The following bits in the response data are used by Linux. The four general characteristics associated with QDIO are in bold.

Bit	Description	Linux Field
12	Dynamic I/O – affects ioctl start request	dynio
41	**QDIO Adapter interrupt facility**	**aif**
45	Multiple channel subsystem facility	mcss
46	Required for channel path description format 1	fcs
48	Extended measurement block support	ext_mb
56	**AIF Time Delay Disablement facility**	**aif_tdd**
58	**QEBSM supported by Channel Subsystem**	**qebsm**
67	**OSA adapter interrupt facility support (or hipersocket)**	**aif_osa**
82	Required for channel path description format 2	cib
88	Fiber channel extensions (transport mode)	fcx

Subchannel Subsystem Call

The Subchannel Subsystem Call initiates a request by the channel subsystem for the specific subchannel identified in the call. The operation code indicates the action the subchannel is to perform.

Request/Response Layout

Disp.	Length	Field	Description
+0	2	req.length	Request length (0x0010, 4064 bytes)
+2	2	req.code	Request code (0x0021)
+4	2	operation_code	Operation code (0, set indicators)
+6	2		reserved
+8	8		reserved
+16	8	summary_indicator_address	Summary indicator absolute address (or zero to reset)
+24	8	subchannel_indicator_address	Subchannel specific indicator absolute address (or zero to reset)
+32	8		Bits 0-3, summary indicator storage key Bits 4-7, subchannel indicator storage key Bits 8-29, reserved, zeros, Bits 30-31, adapter interrupt input/output subclass Bits 32-34, reserved Bit 35, TDD disable bit Bits 36-63, reserved
+40	4		reserved
+44	4	schid	Subchannel for which indicators are to be set
+48	4016		reserved
+4064	2	rsp.length	Response length
+4066	2	rsp.code	Response code
+4070	4		reserved

schid

Bit(s)	Field	Descriptions
0-7	cssid	Channel subsystem ID
8-11		reserved
12	m	
13-14	ssid	Subsystem ID
15	one	Set to one
16-31	sch_no	Subchannel number

Subsystem Queue Description Call

The Subsytem Queue Description Call requests information from one or more subchannels with regard to their support and use of QDIO.

Request/Response Layout

Disp.	Length	Field	Description
+0	2	req.length	Request length (0x0010, 16 bytes)
+2	2	req.code	Request code (0x0024)
+4	2		Bits 0-9 reserved (zeros) Bits 10-11, ssid, channel subsystem id Bits 12-15, fmt, format (0x0)
+6	2	first_sch	First subchannel
+8	2		reserved
+10	2	last_sch	Last subchannel
+12	4		reserved
+16	2	rsp.length	Response length
+18	2	rsp.code	Response code
+20	4		reserved
+24	32	ssqd	Susbsystem Queue Description (SSQD) Response, one per subchannel

Response length for a single subchannel is 40 bytes (8 for the response header, 32 for the SSQD)

Subsystem Queue Description

The SSQD structure is defined in arch/s390/include/asm/qdio.h

Disp.	Length	Field	Description
+0	1	flags	See details below
+1	1		reserved
+2	2	schid	Subchannel Identification
+4	1	qfmt	Queue format, see values below
+5	1	parm	Parm format (Not used by Linux, a guess)
+6	1	qdioac1	Adapter Characteristics, see details below
+7	1	sch_class	Subchannel class
+8	1	pcnt	Port count, number of ports supported by the OSA
+9	1	icnt	Input queues supported (Not used by Linux, a guess)
+10	1		reserved
+11	1	ocnt	Output queue supported (Not used by Linux, a guess)
+12	1		reserved
+13	1	mbccnt	Maximum queue buffer count (Not used by Linux, a guess)
+14	2	qdioac2	Additional access options (Not used by Linux, options unknown)
+16	8	schtoken	Token identifying the subchannel if QEBSM is available
+24	1	mro	not used by Linux, no clue
+25	1	mri	not used by Linux, no clue
+26	1		reserved
+27	1	sbalic	Minimum input SBAL entry count required (Not used by Linux, a guess)
+28	2		reserved
+30	1		reserved
+31	1	mmwc	When not zero, indicates the maximum multiple write buffer count for enhanced SIGA output operations, SIGA function code 3, provided by Hipersocket queues.

flags

This field provides information about QDIO capabilities of a subchannel.

Bit Usage	Linux Symbol	Description
X...	CHSC_FLAG_QDIO_CAPABILITY	Subchannel QDIO capability: 0=incapable, 1=capable
.X..	CHSC_FLAG_VALIDITY	Flag validity: 0=invalid, 1=valid
..XX XXXX		reserved

Subchannels that do not understand how to provide SSQD information will leave this field zero. Subchannels that recognize the request but do not support QDIO must set the flag field to 0x40. Subchannels that are QDIO capable must set the flag field to 0xC0 indicating the information is both valid and QDIO may be enabled. For devices that require QDIO operation, the capability is really from the perspective of the channel subsystem. For such devices, proper device operation requires that QDIO be used.

qdioac1 – Adapter Characteristics

This field provides access to details of the capabilities offered by the adapter's data device. This information influences how the program will interface with the data device for adapter input/output operations.

Bit Usage	Linux Symbol	Description
X...		reserved
.X..	AC1_SIGA_INPUT_NEEDED	SIGA Input needed to retrieve input buffers
..X.	AC1_SIGA_OUTPUT_NEEDED	SIGA Output needed to initiate output
...X	AC1_SIGA_SYNC_NEEDED	SIGA Synchronize needed to update SLSB state
.... X...	AC1_AUTOMATIC_SYNC_ON_THININT	Sync provided automatically on adapter interrrupt by hypervisor
.... .X..	AC1_AUTOMATIC_SYNC_ON_OUT_PCI	Sync provided automatically by hypervisor on PCI from an output queue
.... ..X.	AC1_SC_QEBSM_AVAILABLE	QEBSM available for subchannel
....X	AC1_SC_QEBSM_ENABLED	QEBSM enabled for subchannel

These key values and others related to QDIO are displayed by the Linux kernel when the queues are established with the following message found in cio/qdio_setup.c.

```
qdio: %s %s on SC %x using AI:%d QEBSM:%d PCI:%d TDD:%d SIGA:%s%s%s%s%s%s
```

Inspection of this message from different running system configurations would provide valuable insight into the set of options actually used by these configurations. The settings of these values dictate how the QDIO driver interacts with a specific adapter.

qfmt

Value	Linux Symbol	Description
0x00	QDIO_QETH_QFMT	OSA queue format
0x01	QDIO_ZFCP_QFMT	FCP queue format
0x02	QDIO_IQDIO_QFMT	HiperSocket queue format

QDIO Data Device Commands

The data device associated with a queue direct I/O device is used to provide a communication path between to the QDIO channel subsystem feature and the program. This interface primarily is used to communicate the configuration of the queues the program will utilize.

Code	Channel Command	Description0x
0x1F *	ACTIVATE	Causes the adapter to to initiate I/O operations using the established queues of its data device
0x1B *	ESTABLISH	Informs the QDIO feature of the program's queue configuration for the adapter
0xFA **	READ CONFIGURATION DATA	Returns adapter configuration data
0x04	SENSE	Returns sense data
0xE4	SENSE ID	Return configuration data and Command Information Words

* These commands are specified by means of extended identification Command Information Words supplied by the SENSE ID command and are device dependent. The codes for ACTIVATE and ESTABLISH are identified as the defaults in the Linux code. Any different code consistent with control command codes may be used by the QDIO device for these commands.

** This command code is code identified in IBM manual SA22-7871-01, *z/Architecture Reference Summary*, for a SENSE ID command. No reference to the actual code used by a QDIO data device is available from the Linux source. Linux uses the code provided by a CIW from the device. The QDIO device may support any code as long as it is consistent for a read sense command.

Error Reporting

Linux does not do any detailed analysis of the possible device error status conditions or possible sense bytes. It only logs the reported data for external analysis. It is therefore impossible to determine valid detailed error reporting data for the data device, other than as actually indicated below. Testing with real hardware or with emulated devices under z/VM would be required to determine how such data might be reported. When sense data is present in the IRB, Linux will display the 32-byte Extended-Control Word and 32-byte Extended-Measurement Word in hex.

CLEAR SUBCHANNEL Function

The CLEAR SUBCHANNEL instruction addressed to the data device will remove established queue information. A CLEAR SUBCHANNEL is used following errors encountered by either the ESTABLISH or ACTIVATE commands to reset the QDIO data device.

The CLEAR function is used if the QDIO data device has not been activated by an ACTIVATE command.

HALT SUBCHANNEL Function

The HALT SUBCHANNEL instruction addressed to the data device will cause it to cease handling input or output data. Buffers pending transmission or whose transmission is aborted or suppressed will reflect this with an SLSB state of SLSB_P_INPUT_HALTED or SLSB_P_OUTPUT_HALTED. The QDIO data device is then reset as if a CLEAR function had been issued to the device.

The HALT function is used following successful activation of the device by an ACTIVATE command.

ACTIVATE Command (X'1F')

ACTIVATE command initiates QDIO I/O operations following the establishment of the queues. Activation appears to make active the queues established by the ESTABLISH command and enable the interface between the adapter and the channel subsystem interface for input/output use. It would be at this time that the SBAL structure is likely exposed to the adapter.

The default ACTIVATE command code is X'1F' and the default command length is 0.

The I/O interrupt generated by the completion of the ACTIVATE command is expected to have the PCI channel status set. The ACTIVATE CCW does not require the PCI flag to be set for the PCI status to be returned. The presence of the PCI status indicates successful completion of the ACTIVATE command. Any status not accompanied by the PCI status indicates an ACTIVATE check condition. Linux does not explicitly check for channel end and device end, only PCI on completion of the ACTIVATE command.

QEBSM instructions may return a code that implies an ACTIVATE CHECK condition occurred. A role is implied for ACTIVATE when QEBSM is available. I suspect that ACTIVATE actually validates the data supplied via the ESTABLISH command. Errors encountered during this process may become exposed when the QEBSM instruction is used due to an error state that may occur.

Asynchronous Interrupts

Following successful completion of the ACTIVATE Command, PCI interrupts are used to signal the program of input/operations for which it needs to take action (input present or output complete). Any asynchronous interrupt not including a PCI indication reflect an error

condition within the adapter.

Whether asynchronous interrupts without the PCI indicator being set post activation are actually possible is unclear. However, if such were to occur, Linux would treat them as an error.

ESTABLISH Command (X'1B')

ESTABLISH is a write control command that transfers to the data device the Queue Description Record (see below). The command may also perform a microcode loading/resetting of the adapter. The actual channel command code is provided by a Command Information Word for the command.

A device status of Channel End accompanied by Device End without any other device status indicators indicates that the command executed successfully.

The default ESTABLISH command code is X'1B' and the default command length is 4096.

Errors encountered during the ESTBLISH command will be reflected in either the channel status or device status. A successful ESTABLISH command will be indicated by a device end unaccompanied by any other device status condition.

Linux does not do an analysis of the possible device error status conditions nor any sense bytes. It only logs the reported data for external analysis. It is therefore impossible to determine valid error reporting data for the data device

Queue Description Record (QDR)

The QDR is written to an adapter's data device subchannel to establish the queues. The Command Information Word for the ESTABLISH channel command is used for this write operation.

Disp.	Length	Field	Description
+0	1	qfmt	Queue format (see SSQD qfmt field)
+1	1	pfmt	Parameter format
+2	1		reserved
+3	1	ac	Adapter characteristics
+4	1		reserved
+5	1	iqdcnt	Number of input queue descriptors
+6	1		reserved
+7	1	oqdcnt	Number of output queue descriptors
+8	1		reserved
+9	1	iqdsz	Size of an Input Queue Descriptor in 4-byte words, 8 words

Disp.	Length	Field	Description
+10	1		reserved
+11	1	oqdsz	Size of an Output Queue Descriptor in 4-byte words, 8 words
+12	4		reserved
+16	32		reserved
+48	8	qiba *	Queue Information Block (QIB) absolute address
+56	4		reserved
+60	1	qkey	QIB storage key, bits 0-3, reserved, bits 4-7
+61	3		reserved
+64	4032	qdf0s	Space for 126 Format-0 Queue Descriptors, input queues followed by output queues.

* In ESA/390, bits 0-31 of an absolute address are reserved.

Queue Descriptor, Format 0 (QDF0)

A format 0 Queue Descriptor describes attributes of a single queue.

Disp.	Length	Field	Description
+0	8	sliba *	Storage List Information Block absolute address
+8	8	sla *	Storage List absolute address
+16	8	slsba *	Storage List State Block absolute address
+24	4		reserved
+28	2	skeys	Storage keys (see bit usage below)
+30	2		reserved

* In ESA/390, bits 0-31 of an absolute address are reserved.

skeys

Bit Usage	Linux Symbol	Description
XXXX	akey	Storage List Information Block (SLIB) storage key
. . . . XXXX	bkey	Storage List (SL) storage key
. . . . XXXX	ckey	Storage Block Address List (SBAL) and storage buffer storage key

Bit Usage	Linux Symbol	Description
. . . . XXXX	dkey	Storage List State Block (SLSB) storage key

Queue Information Block (QIB)

Alignment: 256

Storage Key: QDR, qkey field

Disp.	Length	Field	Description
+0	1	qfmt	Queue format (see SSQD qfmt field)
+1	1	pfmt	Parameter format, 0 used by QETH and ZFCP
+2	1	rflags	Bit 0 set to 1 enables QEBSM on the queue.
+3	1	ac	Adapter characteristics, bit 1 set to 1 indicates outbound PCI supported by the program.
+4	4		reserved
+8	8	isliba	Absolute address of the first input SLIB in the linked list of input SLIB's
+16	8	osliba	Absolute address of the first output SLIB in the linked list of output SLIB's
+24	8		reserved
+32	8	ebcnam	Adapter identifier in EBCDIC
+40	88		reserved
+128	128	parm	128-byte implementation dependent parameters

Network Adapter Parameter Data

Parameters specific to the adapter's operation may be provided in the “parm” field of the QIB. This data is specific to the device. Linux refers to this data as “impl” data. The network adapter accepts two parameters. Each parameter consists of four four-byte words. The first word consists of a four-character EBCDIC sequence identifying the parameter, the parameter id. Each of the remaining three words contain a binary value in the word, parameter values 0-2. The use of EBCDIC character sequence to identify the parameter data suggests that any sequence of parameters is legitimate, although Linux first specifies the PCI Threshold Parameter followed by the Block Timer Parameter.

PCI Threshold Parameter – 'PCIT'

The PCI Threshold parameter defines a set of PCI related thresholds. The parameter id is the EBCDIC character sequence 'PCIT' (0xD7, 0xC4, 0xC9, 0xE3).

Value 0 – Number of physical inbound buffers

Linux attribute: /sys/bus/ccwgroup/drivers/qeth/<devise_bus_id>/buffer_count

These are the number of actual buffers for input to be used by Linux. As Linux marches around an input queue buffer ring, these buffers will be reused as they become available. When the adapter has used this number of buffers, it must present a PCI or adapter interrupt to the program

Value 1 – Set to zero by Linux. Meaning uncertain, but considering the context, it could be the threshold of output buffers after whose transmission the adapter should generate an output PCI. Zero could mean none implying the program will set the PCI flag in an SBAL to trigger outbound PCI's (which Linux does in fact do.)

Value 2 – Set to a value of 3 by Linux. Purpose unknown.

Block Timer Parameter – 'BLKT'

The Block Timer parameter defines a set of timer values to be used by the adapter. The parameter id is the EBCDIC character sequence 'BLKT' (0xC2, 0xD3, 0xD2, 0xE3). The BLKT parameter describes how the adapter should utilize the time delay facility. It is this facility that can be disabled by the TDD Facility when Adapter Interrupts are used.

The values are times. The time unit is not defined, but it is assumed to be milliseconds.

Value 0 – Total time that the adapter should delay from receipt of a frame before generating a PCI interrupt. Legitimate values accepted by Linux are in the range of 0-1000 inclusive.

Linux attribute: /sys/bus/ccwgroup/drivers/qeth/<devise_bus_id>/total

Value 1 – Maximum time between receipt of network frames before generating a PCI interrupt. Legitimate values accepted by Linux are in the range of 0-100 inclusive.

Linux attribute: /sys/bus/ccwgroup/drivers/qeth/<devise_bus_id>/total

Value 2 – Maximum time between receipt of network jumbo frames before generating a PCI interrupt. Legitimate values accepted by Linux are in the range of 0-100 inclusive.

Linux attribute: /sys/bus/ccwgroup/drivers/qeth/<devise_bus_id>/jumbo

Storage List Information Block (SLIB)

Alignment: 2048

Storage Key: QDR, Queue Descriptor, akey field

Disp.	Length	Field	Description
+0	8	nsliba	Next SLIB absolute address
+8	8	sla	Storage List absolute address
+16	8	slsba	Storage List State Block absolute address
+24	1000		reserved
+1024	1024	slibe	128 8-byte implementation dependent parameters, 1 per the 128 maximum buffers per queue

Storage List (SL)

Alignment: 1024

Storage Key: QDR, Queue Descriptor, bkey field

A Storage List contains 128 8-byte storage list elements, one element per storage buffer. Each Storage List element contains the absolute address of a Storage Block Address List. In ESA/390 mode, bits 0-31 of the storage list element is reserved.

Storage Block Address List (SBAL)

Alignment: 256

Storage Key: QDR, Queue Descriptor, ckey field

The Storage Block Address List contains a maximum of 16 Storage Block Address List Entries. The entire list corresponds to the description of a single queue buffer. A single SBAL may consist of one or more SBAL entires. FCP uses 16 entries per buffer. QETH tends to use one per buffer.

Storage Block Address List Entry (SBALE)

Alignment: 16

Storage Key: QDR, Queue Descriptor, ckey field

The key to be used to access a buffer is nowhere specified in the Linux structures. The assumption is made that the same key used for the SBAL is used to access system buffers. It is also possible that one of the reserved fields of the SBALE is actually used to specify the system buffer key and that Linux has simply not documented its usage. Only testing with multiple implementations that use QDIO or real hardware could validate this assumption.

Disp.	Length	Field	Description
+0	4	flags	Flags associated with this entry
+4	4	length	Length of the storage area associated with this entry

Disp.	Length	Field	Description
+8	8	addr *	Absolute address of the storage area associated with this entry

* In ESA/390 mode bits 0-31 are reserved.

flags

When output packing mode is used, multiple adapter frames or packets will be identified by a single SBAL. Each packet or frame will have a sequence of first, middle and last fragment settings as required for the packet.

Bit 25 is set to cause an output PCI to be triggered on completion of the output operation initiated by a SIGNAL ADAPTER Output function. This is normally set in the first SBAL entry of the first buffer being sent to the adapter.

Bits 0-7 Usage	Linux Symbol	Description
X...		reserved
.1..	SBAL_FLAGS_LAST_ENTRY	Last entry in the SBAL, terminates processing of this SBAL by the data device or adapter
..1.	SBAL_FLAGS_CONTIGUOUS	This entry is contiguous to the previous
...X		reserved
.... 01..	SBAL_FLAGS_FIRST_FRAG	This entry is the first fragment of of a multi-fragment buffer
.... 10..	SBAL_FLAGS_MIDDLE_FRAG	This entry is a middle fragment of the buffer
.... 11..	SBAL_FLAGS_LAST_FRAG	This entry is the last or only fragment of the buffer
.... ..XX		reserved

Bits 8-23 are reserved.

Bits 24-31 Usage	Linux Symbol	Description
X...		reserved
.1..		Set by the program to request PCI upon completion of the handling of this output queue buffer.
..xx xxxx		reserved

Bits 24-31 are also used in entries 14 and 15 to reflect error conditions reported by the the presence of the error state in the SLSB.

The usage described above is generic. Zfcp extends use of the flags field. Zfcp extensions are not described here.

Storage List State Block (SLSB)

Alignment: 256

Storage Key: QDR, Queue Descriptor, dkey field

The Storage List State Block contains 128 contiguous one-byte state indicators describing the current state of each of the 128 possible buffers in the Storage List, hence the name, Storage List State Block. There is a one-to-one correspondence between the a Storage List entry, a pointer to a single Storage Block Address List, and the state indicator in the Storage List State Block.

Contains the buffer state of each buffer, one entry for each QDIO buffer. Each entry is a single byte in length. The list must be 256-byte aligned. Each state entry is 8-bits in length with the following assignments

Bits 0,1 – Buffer owner (a bits)

Bit 2 – Buffer type (b bit)

Bit 3 – validity bit (v bit)

Bits 4-7 – State (y bits)

State Bits	Linux Name	Description
aabv 0000	SLSB_STATE_NOT_INIT	Not initialized (initial state)
aabv 0001	SLSB_STATE_EMPTY	Buffer is not eligible to swap ownership
aabv 0010	SLSB_STATE_PRIMED	Buffer eligible to swap ownership with data
aabv 1110	SLSB_STATE_HALTED	I/O halted
aabv 1111	SLSB_STATE_ERROR	Error occurred during I/O
aa0v yyyy	SLSB_TYPE_INPUT	Input buffer
aa1v yyyy	SLSB_TYPE_OUTPUT	Output buffer
10xv yyyy	SLSB_OWNER_PROG	Program owns the buffer
01xv yyyy	SLSB_OWNER_CU	Control unit owns the buffer
1111 1111	SLSB_ERROR_DURING_LOOKUP	(not referenced by program)

Linux does not use the above bit definitions directly, but rather always combines them into a single state. These combined states therefore represent the buffer conditions that Linux recognizes. The recognized buffer states and usage are described in the following table.

State Setting	Linux State Name	Linux State Usage
1000 0000 (0x80)	SLSB_P_INPUT_NOT_INIT	Set: qdio_main.c/qdio_init_buf_states Set: qdio_main.c/qdio_stop_polling

State Setting	Linux State Name	Linux State Usage
		Set: qdio_main.c/inboud_primed (no QEBSM) Ref: qdio_main.c/get_inbound_buffer_frontier Ref: qdio_debug.c/qstat_show
1000 0001 (0x81)	SLSB_P_INPUT_ACK	Set: qdio_main.c/inboud_primed (no QEBSM) Ref: qdio_main.c/get_inbound_buffer_frontier Ref: qdio_debug.c/qstat_show
0100 0001 (0x41)	SLSB_CU_INPUT_EMPTY	Ref: qdio_main.c/get_inbound_buffer_frontier Set: qdio_main.c/handle_inbound Ref: qdio_debug.c/qstat_show
1000 0010 (0x82)	SLSB_P_INPUT_PRIMED	Ref: qdio_main.c/get_inbound_buffer_frontier Ref: qdio_main.c/qdio_inbound_q_done Ref: qdio_debug.c/qstat_show
1000 1110 (0x8E)	SLSB_P_INPUT_HALTED	Ref: qdio_debug.c/qstat_show
1000 1111 (0x8F)	SLSB_P_INPUT_ERROR	Ref: qdio_main.c/get_inbound_buffer_frontier Ref: qdio_debug.c/qstat_show
1010 0000 (0xA0)	SLSB_P_OUTPUT_NOT_INIT	Set: qdio_main.c/qdio_init_buf_states Ref: qdio_main.c/get_outbound_ buffer_frontier Ref: qdio_debug.c/qstat_show
1010 0001 (0xA1)	SLSB_P_OUTPUT_EMPTY	Ref: qdio_main.c/get_outbound_ buffer_frontier Ref: qdio_debug.c/qstat_show
0110 0010 (0x62)	SLSB_CU_OUTPUT_PRIMED	Ref: qdio_main.c/get_outbound_ buffer_frontier Set: qdio_main.c/handle_outbound Ref: qdio_debug.c/qstat_show
1010 1110 (0xAE)	SLSB_P_OUTPUT_HALTED	Ref: qdio_main.c/get_outbound_ buffer_frontier Ref: qdio_debug.c/qstat_show
1010 1111 (0xAF)	SLSB_P_OUTPUT_ERROR	Ref: qdio_main.c/get_outbound_ buffer_frontier Ref: qdio_debug.c/qstat_show
1111 1111 (0xFF)	SLSB_ERROR_DURING_LOOKUP	Storage reference errors – access exceptions

READ CONFIGURATION DATA Command (0xFA)

Returns adapter configuration information. Linux utilizes 64 bytes of the potentially supplied information. The minimum configuration record supported is 96-bytes, containing an Emulation NED and I/0 Device NED and a specific NEQ.

The Linux QETH device driver uses the READ CONFIGURATION DATA command addressed to the adapter group device's data subchannel to determine some elements of the adapter's physical or emulated configuration. It is by use of the READ CONFIGURATION DATA information that the QETH driver is able to determine if the adapter is a physical OSA Express adapter or a z/VM Guest LAN adapter.

The configuration record includes

Bytes 0-31 – Emulation Node Element Descriptor (NED)

Bytes 32-63 – I/O Device Node Element Descriptor (NED)

Bytes 64-95 – Node Element Qualifier (NEQ)

Refer to IBM manual SA22-7204-01, *Enterprise Systems Architecture/390 Common I/O Device Commands*, for NED and NEQ structure layouts. Detailed below are specific uses of the node element structures use by the Linux QETH drivers and supplied by the network adapter QDIO data device.

Emulation Node Element Descriptor

Bytes 10,11 of the Emulation NED, the Plant of Manufacture field, contain in EBCDIC, the characters 'VM' when the adapter is a z/VM Guest LAN.

Bytes 30,31 of the Emulation NED, the tag field, contain, the CHPID in byte 30 and the unit address in byte 31.

I/O Device Node Element Descriptor

Byte 31 of the I/O Device Node Element Descriptor, the second byte of the tag field, contains the Control Unit Logical Address.

Node Element Qualifier

The Node Element Qualifier is not referenced by Linux.

SENSE Command (0x04)

The SENSE command provides device specific sense data. SENSE data generated by the QDIO device is displayed, but not analyzed by Linux. The content and nature of the information can not be determined. Linux does not typically issue a SENSE command. Linux relies upon concurrent sense being set in the subchannel's Path Management Control Word. A maximum of 32-bytes of sense data may be stored in the Extended-Control Word of the Interrupt-Response Block.

SENSE ID Command (0xE4)

A QDIO data device provides both basic and extended identification information in the data returned by SENSE ID. The QDIO data device will return 20-bytes of data:

8 bytes of basic identification information

12 bytes of extended identification information from three 4-byte CIW's.

Basic Identification Information

Control unit and device model information is provided by the basic identification information. The following basic identification information is provided by QDIO devices.

Device	Validity	CU Type	CU Model	Device Type	Device Model	Extended
Bytes	**0**	**1,2**	**3**	**4,5**	**6**	**7**
OSA Express	0xFF	0x1731	0x01	0x1732	0x01	0x00
SCSI	0xFF	0x1731	0x03	0x1732	0x03	0x00
Privileged SCSI	0xFF	0x1731	0x03	0x1732	0x04	0x00
Hipersocket	0xFF	0x1731	0x05	0x1732	0x05	0x00
OSA for NCP	0xFF	0x1731	0x06	0x1732	0x06	0x00

Extended Identification Information

The extended identification information immediately follows the basic identification information.

The extended identification information takes the form of Command Information Words. A Command Information Word (CIW) provides channel command information. Each 32-bit CIW has the following format.

Bits	Field	Description
0,1	Entry Type	Bit zero is set to 0, bit one is set to 1.
2,3	Reserved	All bits set to zero
4-7	Command Type	0x0 = QDIO Read Configuration Data command 0x3 = QDIO Establish command 0x4 = QDIO Activate command
8-15	Command	0xFA = QDIO Read Configuration Data Command 0x1B = QDIO Establish command 0x1F = QDIO Activate command
16-31	Command Count	Read Configuration Data command = 96 Establish command = 4096 Activate command = 0

CPU Instructions

Three CPU instructions are related to the QDIO feature:

EXTRACT QUEUE BUFFER STATE provided by the QEBSM facility,

SIGNAL ADAPTER provided by the base QDIO feature,

SET QUEUE BUFFER STATE provided by QEBSM facility, and

SUBSYSTEM VARY STATUS provided by AIF.

EQBS – EXTRACT QUEUE BUFFER STATE

```
EQBS R1,R2,R3,M4
```

0xB99C	R3	M4	R1	R2

EXTRACT QUEUE BUFFER STATE performs in hardware what had been performed previously in software by the qdio_do_eqbs function qdio_main.c provided in the “Usage” section.

This instruction requires the QEBSM facility to be provided by the Channel Subsystem. This facility is indicated in the general channel subsystem characteristics data by bit 58 being set to 1. This facility is only available with z/Architecture.

If QEBSM is neither available nor enabled for the adapter SQBS will generate an operation exception program interrupt. The state of QEBSM for a specific adapter can be determined from the Subsystem Queue Description provided by the CHSC targeted to the adapter's data device subchannel.

The SLSB that is being targeted by the instruction is identified not by its storage address but by providing the:

subchannel token (see Subchannel Queue Description) in the operand 3's odd register and

queue index number in operand 1 (specified by the program in the Queue Descriptor Record).

The address of the targeted SLSB is then determined based upon the information provided in the Queue Descriptor Record, SLIB and QIB of the targeted queue.

EQBS is an interruptible instruction.

At the start of the instruction:

Operand 1, bits 0-31 contains the starting buffer queue index number and bits 32-63 the starting SLSB buffer number to be set.

The queue index number corresponds to the index of the queue as specified in the ESTABLISH queue data record. Input queues are numbered from 0 to number of input queues minus 1 and output queues are numbered from the number of input queues to the number of output queues plus the number of input queues minus 1.

If “in.qs” is the number of input queues and “out.qs” are the number of output queues specified in the Queue Data Record:

input queue numbers are: 0 <= input_queue_number < in.qs and

output queue numbers are: in.qs <= output_queue_number < (in.qs + out.qs)

Operand 2, contains in bit 0 an indicator of whether input buffers are to be automatically acknowledged. When bit 0 is 1, input buffers are automatically acknowledged, otherwise they are not.

Auto acknowledgment converts buffer states of SLSB_P_INPUT_PRIMED (0x82) to SLSB_P_INPUT_ACK (0x81). Technically this converts a buffer that is input primed and owned by the program (the adapter has provided input to the program) to a program owned empty buffer. The buffer is not actually “empty” because the control unit does not own it for new input. Only when the “empty” buffer is transferred back to control unit ownership is the buffer truly available for new input and can then be considered empty.

Operand 3 is an even/odd pair of registers.

The even register of the pair contains in bits 32-63 the number of buffers, starting with the buffer identified by operand 1, whose state is desired to be examined.

The odd register of the pair contains the 64-bit subchannel token used to identify the queue. The token is provided to the program my means of the Subchannel Queue Descriptor information by means of the CHANNEL SUBSYSTEM CALL issued to the channel subsystem.

The source of the token in Linux is: `irq_ptr->ssqd_desc.sch_token`.

Operand 4 is set to zero by Linux. If the mask has a role and what it might be is impossible to determine by Linux usage of the instruction.

At the conclusion of the instruction:

Operand 1, bits 0-31, contains the buffer number of the next buffer state to be extracted.

Operand 2 contains in bits 56-63 the extracted state of the examined buffers.

Operand 3's even/odd pair of registers :

The even register

bits 0-31, contains a return code and

bits 32-64 contains the number of buffers states not examined as

requested. The count provided at the start of the instruction is decremented for each successfully examined buffer states.

The odd register is unchanged.

Return Code Values (bits 0-31 of operand three's even register):

0x00, 0 - all buffer states successfully processed.

0x20, 32 – all buffer states successfully processed and next buffer state different

0x60, 96 – not all buffers processed

0x61, 97 – not all buffers processed (The meaning of the low order bit being set is not able to be determined from the Linux code.)

otherwise, an ACTIVATE CHECK CONDITION has occurred for SQBS.

Example usage: drivers/s390/cio/qdio.h

```
/*  call in qdio_main.c           prototype in qdio.h  */
do_eqbs(q->irq_ptr->sch_token,  /* u64 token            */
        state,                  /* unsigned char *state */
        nr,                     /* int queue            */
        &tmp_start,             /* int *start           */
        &tmp_count,             /* int *count           */
        auto_ack);              /* int ack              */

static inline int do_eqbs(u64 token, unsigned char *state, int queue,
                          int *start, int *count, int ack)
{
        register unsigned long _ccq asm ("0") = *count;
        register unsigned long _token asm ("1") = token;
        unsigned long _queuestart = ((unsigned long)queue << 32) | *start;
        unsigned long _state = (unsigned long)ack << 63;

        asm volatile( /*        RRF3 format    R1 R2 R3M4 */
                "       .insn   rrf,0xB99c0000,%1,%2,0,0"
        /* Output: */
                : /* %0 */  "+d" (_ccq),          /* %r0 */
                  /* %1 */  "+d" (_queuestart),   /* compiler selected, R1 */
                  /* %2 */  "+d" (_state)         /* compiler selected, R2 */
        /* Input   */
                : /* %3 */  "d" (_token)          /* %r1 */
        /* Clobber */
                : "memory", "cc");
        *count = _ccq & 0xff;
        *start = _queuestart & 0xff;
        *state = _state & 0xff;

        return (_ccq >> 32) & 0xff;
}
```

Usage

drivers/s390/cio/qdio_main.c

```
/**
 * qdio_do_eqbs - extract buffer states for QEBSM
 * @q: queue to manipulate
 * @state: state of the extracted buffers
 * @start: buffer number to start at
 * @count: count of buffers to examine
 * @auto_ack: automatically acknowledge buffers
 *
 * Returns the number of successfully extracted equal buffer states.
 * Stops processing if a state is different from the last buffers state.
 */
static int qdio_do_eqbs(struct qdio_q *q, unsigned char *state,
                        int start, int count, int auto_ack)
{
        unsigned int ccq = 0;
        int tmp_count = count, tmp_start = start;
        int nr = q->nr;
        int rc;

        BUG_ON(!q->irq_ptr->sch_token);
        qdio_perf_stat_inc(&perf_stats.debug_eqbs_all);

        if (!q->is_input_q)
                nr += q->irq_ptr->nr_input_qs;
again:
        ccq = do_eqbs(q->irq_ptr->sch_token, state, nr, &tmp_start, &tmp_count,
                      auto_ack);
        rc = qdio_check_ccq(q, ccq);

        /* At least one buffer was processed, return and extract the remaining
         * buffers later.
         */
        if ((ccq == 96) && (count != tmp_count)) {
                qdio_perf_stat_inc(&perf_stats.debug_eqbs_incomplete);
                return (count - tmp_count);
        }

        if (rc == 1) {
                DBF_DEV_EVENT(DBF_WARN, q->irq_ptr, "EQBS again:%2d", ccq);
                goto again;
        }

        if (rc < 0) {
                DBF_ERROR("%4x EQBS ERROR", SCH_NO(q));
                DBF_ERROR("%3d%3d%2d", count, tmp_count, nr);
                q->handler(q->irq_ptr->cdev,
                           QDIO_ERROR_ACTIVATE_CHECK_CONDITION,
                           0, -1, -1, q->irq_ptr->int_parm);
                return 0;
        }
        return count - tmp_count;
```

```
}
```

EQBS is used in the generic function qdio_main.c/get_buf_state. get_buf_state will call get_buf_states which will determine if the program can use EQBS or extract the information itself. Where get_buf_state or get_buf_states are used implies the situations in which EQBS is used.

SIGA – SIGNAL ADAPTER

```
SIGA D2(B2)
```

0xB274	B2	D2

SIGNAL ADAPTER requests a QDIO interface to perform various functions. Arguments are provided to the adapter being signaled in general registers. The results of the instruction are indicated by setting of the condition code.

The following arguments are provided in assigned registers:

General Register 0 – the function being signaled to the adapter and method of adapter identification:

Bits 0-55 are set to 0

Bit 56 is set to

0 if General Register 1 contains a subchannel identification, or

1 if General Register 1 contains a subchannel token.

A subchannel token is only available if QEBSM is available. QEBSM is restricted to z/Architecture mode.

Bits 57-64 identifies the SIGNAL ADAPTER function being signaled:

0 = Standard Output function (single packet of frame per queue buffer, SBAL)

1 = Input function

2 = Synchronize function

3 = Enhanced Output function (multiple packets or frames per queue buffer, SBAL). Use of multiple packets or frames is referred to as “packing mode”.

Linux only uses the subchannel token option with the output or enhanced output functions. It should be presumed to be available for the other functions as well.

General Register 1 – the subchannel identification or, if QEBSM is available and enabled, the subchannel token of the adapter being signaled. Register 0, bit 56, will be set to 1 if general register 1 contains a subchannel token.

General Register 2 – an input or output queue mask, depending upon the function requested.

General Register 3 – an input queue mask when both an input and output queue mask are required, the output mask being contained in register 2.

Notes on Linux usage: Operand 2 is always 0 when used by Linux. This results in SIGA 0(0), that is, both the base and displacement are zero in the machine instruction. It is therefore impossible to determine if operand 2 has any use. The above behavior suggests that operand 2 is actually ignored by SIGNAL ADAPTER. If the second operand is used, it likely points to a structure that contains information comparable to that supplied in general registers 0-3 as described above.

Queue Mask

The queue mask is a sequence of 32-bits numbered from most-significant bit to least starting with with zero and ending with 31. Each numbered bit corresponds to an established queue described by a Queue Description record. The queue mask therefore identifies one or more queues targeted by a function. Bits corresponding to queues that have not been established are ignored.

Signal Synchronize (Function 2)

The Synchronize Function updates either the adapter's SLSB or program's SLSB with the other's state block information. Buffer states in the program's SLSB indicating control unit ownership will cause the adapter's SLSB to be updated for that buffer. Buffers in the adapter's SLSB indicating ownership by the program will cause the program's SLSB to be updated.

Programming Note

The signal synchronize function is required during output operations before determining output buffer completion status when:

OSA or FiberChannel adapters do not support PCI output interrupts or

a multicast queue is used with a Hipersocket adapter.

The signal synchronize function is required for anytime the program needs to inform the adapter of SLSB state change or needs to update its representation of the state maintained by the adapter.

Example Usage: drivers/s390/cio/qdio_main.c

```
static inline int do_siga_sync(struct subchannel_id schid,
                               unsigned int out_mask, unsigned int in_mask)
{
      register unsigned long __fc asm ("0") = 2;
      register struct subchannel_id __schid asm ("1") = schid;
      register unsigned long out asm ("2") = out_mask;
      register unsigned long in asm ("3") = in_mask;
      int cc;

      asm volatile(
```

```
            "     siga  0\n"
            "     ipm   %0\n"
            "     srl   %0,28\n"
        /* Output: */
            : /* %0 */ "=d" (cc)      /* Compiler selected, IPM R1 */
        /* Input: */
            : /* %1 */ "d" (__fc),    /* %r0 */
              /* %2 */ "d" (__schid), /* %r1 */
              /* %3 */ "d" (out),     /* %r2 */
              /* %3 */ "d" (in)       /* %r3 */
        /* Clobber: */
            : "cc" );
        return cc;
}
```

Signal Input (Function 1)

Output operations are initiated for the input queue identified by the queue mask in general register 2 associated with the data device associated with the subchannel identified by register 1.

The input operations will cause data to be placed in the areas identified by the SBAL associated with SLSB's in the SLSB_CU_INPUT_EMPTY state. Upon completion of the transfer, each SBAL data area will contain one or more fragments of a single inbound unit of adapter data.

The Input function has an implied SIGNAL ADAPTER Synchronize function, updating the adapter with the now available input buffers.

If inbound packing mode is used, the QETH header must reside completely in a single SBALE storage area. It may not span in the storage areas of two SBALE's.

Note: Linux inbound processing tends to rely heavily on the packet contents for length, not the SBALE data. More analysis is required to determine if all of the output uses of SBALE's are found on the input side. It is reasonable to assume it is possible. However, no requirement appears to exist requiring an emulation adapter to support all of the options that appear in the output case for input.

Condition Codes:

0 = successful execution

1 = should not occur, is an error

2 = subchannel busy or adapter busy

3 = an error occured.

Example usage:

```
static inline int do_siga_input(struct subchannel_id schid, unsigned int
mask)
{
```

```
        register unsigned long __fc asm ("0") = 1;
        register struct subchannel_id __schid asm ("1") = schid;
        register unsigned long __mask asm ("2") = mask;
        int cc;

        asm volatile(
              "     siga  0\n"
              "     ipm   %0\n"
              "     srl   %0,28\n"
        /* Output: */
              : /* %0 */ "=d" (cc)
        /* Input: */
              : /* %1 */ "d" (__fc),
                /* %2 */ "d" (__schid),
                /* %3 */ "d" (__mask)
        /* Clobber: */
              : "cc", "memory" );
        return cc;
}
```

Signal Output (Function 0) and Signal Enhanced Output (Function 3)

Output operations are initiated for the output queue identified by the queue mask in general register 2 associated with the data device associated with the subchannel or the subchannel token (if Register 0, bit 56 is set to 1) identified by register 1.

The output operation starts with the next buffer in the queue to be processed by the adapter if it has been primed for output and ownership has been passed to the adapter.

General register 0 contains either 0 if the operation is for one frame or packet per buffer or 3 if the operation is for multiple frames or packets. Multiple frames or packets requires the mmwc queue description field to be greater than 1. Use of multiple frames or packets per queue buffer is referred to as “packing mode”.

Following execution of the instruction general register bit 0 in ESA/390 mode, or bit 32, in z/Architecture mode, will be set to one if the adapter is busy and unable to process output buffers, zero otherwise. All other register bits are set to zero at completion of the instruction.

General register 1 contains the subchannel of the data device being addressed

General register 2 contains the output queue mask of the output queue being targeted for output.

Following execution of the instruction bit 0, in ESA/390 mode, or bit 32, in z/Architecture mode, will be set to one if the adapter is busy and unable to process output buffers, zero otherwise. All other register bits of general register zero are set to zero at completion of the instruction. The busy indication will be provided only in conjunction with condition code 2 being set.

Exceptions

Access exceptions may be recognized while accessing buffer data.

Condition Codes:

0 = successful execution

1 = should not occur, is an error

2 = subchannel busy or adapter busy

3 = an error occured.

Programming Note

When a Hipersocket adapter sets the busy indication in general register 0 in conjunction with condition code 2, the program should take down the queue. Guest LAN's may also set this condition during reconfiguration. In the latter case, the queues should be taken down only if the condition persists (see Linux 2.4.37.9, drivers/s390/qdio.c).

Example usage:

```
static inline int do_siga_output(unsigned long schid, unsigned long mask,
                         unsigned int *bb, unsigned int fc)
{
      register unsigned long __fc asm("0") = fc;
      register unsigned long __schid asm("1") = schid;
      register unsigned long __mask asm("2") = mask;
      int cc = QDIO_ERROR_SIGA_ACCESS_EXCEPTION;

      asm volatile(
            "     siga  0\n"
            "0:   ipm   %0\n"
            "     srl   %0,28\n"
            "1:\n"
            EX_TABLE(0b, 1b)
            : "+d" (cc), "+d" (__fc), "+d" (__schid), "+d" (__mask)
            : : "cc", "memory");
      *bb = ((unsigned int) __fc) >> 31;
      return cc;
}
```

SQBS – SET QUEUE BUFFER STATE

SQBS R1,R3,D2(B2)

0xEB	R1	R3	B2	DL2	DH2	0x8A

SET QUEUE BUFFER STATE performs in hardware what had been performed previously in software by the qdio_do_sqbs function qdio_main.c provided in the “Usage” section.

This instruction requires the QEBSM facility to be provided by the Channel Subsystem. This facility is indicated in the general channel subsystem characteristics data by bit 58 being set to 1. This facility is only available with z/Architecture.

If QEBSM is neither available nor enabled for the adapter SQBS will generate an operation exception program interrupt. The state of QEBSM for a specific adapter can be determined

from the Subsystem Queue Description provided by the CHSC targeted to the adapter's data device subchannel.

The SLSB that is being targeted by the instruction is identified not by its storage address but by providing the:

subchannel token (see Subchannel Queue Description) in the operand 3 odd register and

queue index number in operand 1 (specified by the program in the Queue Descriptor Record).

The address of the targeted SLSB is then determined based upon the information provided in the adapter's Queue Descriptor Record, SLIB and QIB of the targeted queue.

At the start of the instruction:

Operand 1, bits 0-31 contains the starting buffer queue index number and bits 32-63 the starting SLSB buffer number to be set.

The queue index number corresponds to the index of the queue as specified in the ESTABLISH queue data record. Input queues are numbered from 0 to number of input queues minus 1 and output queues are numbered from the number of input queues to the number of output queues plus the number of input queues minus 1.

If “in.qs” is the number of input queues and “out.qs” are the number of output queues specified in the queue data record:

input queue numbers are: 0 <= input_queue_number < in.qs and

output queue numbers are: in.qs <= output_queue_number < (in.qs + out.qs)

Operand 2, rather than being used as an address, contains the state to which the queue buffer states are to be changed. The queue is identified by Operand 1 and the even register of the Operand 3 pair.

Operand 3 is an even/odd pair of registers.

The even register of the pair contains the number of buffers, starting with the buffer identified by operand 1, whose state is desired to be changed.

The odd register of the pair contains the 64-bit queue token used to identify the queue. The source of the token is: `irq_ptr->ssqd_desc.sch_token`

Upon completion of the instruction:

Operand 1 contains in bits 56-63 the index of the first queue whose state was not changed.

Operand 3's even register contains:

a return code in bits 0-31, see qdio_check_ccq() for meaning, and

the number of buffers whose states were not inspected for change of the total originally requested in bits 32-63.

Example usage: drivers/s390/cio/qdio.h

```
static inline int do_sqbs(u64 token, unsigned char state, int queue,
                          int *start, int *count)
{
       /* Rx output %r0 will contain the count */
       register unsigned long _ccq asm ("0") = *count;
       /* Input %r1 contains the token */
       register unsigned long _token asm ("1") = token;

       unsigned long _queuestart = ((unsigned long)queue << 32) | *start;


       asm volatile(/*                          R1R3 D2B2
             “       .insn rsy,0xeb000000008A,%1,0,0(%2)”
       /* Output: */
             : /* %0 */  “+d” (_ccq),                 /* %r0 */
               /* %1 */  “+d” (_queuestart)           /* Compiler selected, R1 */
       /* Input:  */
             : /* %2 */  “d” ((unsigned long)state),  /* Compiler selected, B2 */
               /* %3 */  “d” (_token)                 /* %r1 */
       /* Clobber */
             : “memory”, “cc” );

       *count = _ccq & 0xff;
       *start = _queuestart & 0xff;
       return (_ccq >> 32) & 0xff;  /* Returns the return code */
```

The functionality of SQBS is provided by the function set_buf_states in qdio_main.c. This function uses SQBS if QEBSM is available or performs the following logic if it is not present:

```
       for (i = 0; i < count; i++) {
            xchg(&q->slsb.val[bufnr], state);
            bufnr = next_buf(bufnr);
       }
       return count;
```

xchg atomically sets state of the buffer in the SLSB. next_buf will increment the buffer number, wrapping back to zero if the count goes beyond the last buffer. This code snippet defines the operation of the SQBS instruction.

SQBS may be interrupted. As each buffer is processed the count of buffers to change (in the even register of the operand 3 pair) is decremented by one and the buffer number is incremented by one, wrapping at the end of the queue's buffer. The instruction may be executed with the updated values to continue execution until all of the desired buffers have been changed.

Usage

drivers/390/cio/qdio_main.c

```
/**
 * qdio_do_sqbs - set buffer states for QEBSM
 * @q: queue to manipulate
 * @state: new state of the buffers
 * @start: first buffer number to change
 * @count: how many buffers to change
 *
 * Returns the number of successfully changed buffers.
 * Does retrying until the specified count of buffer states is set or an
 * error occurs.
 */
static int qdio_do_sqbs(struct qdio_q *q, unsigned char state, int start,
                        int count)
{
        unsigned int ccq = 0;
        int tmp_count = count, tmp_start = start;
        int nr = q->nr;
        int rc;

        if (!count)
                return 0;

        BUG_ON(!q->irq_ptr->sch_token);
        qdio_perf_stat_inc(&perf_stats.debug_sqbs_all);

        if (!q->is_input_q)
                nr += q->irq_ptr->nr_input_qs;
again:
        ccq = do_sqbs(q->irq_ptr->sch_token, state, nr, &tmp_start, &tmp_count);
        rc = qdio_check_ccq(q, ccq);
        if (rc == 1) {
                DBF_DEV_EVENT(DBF_INFO, q->irq_ptr, "SQBS again:%2d", ccq);
                qdio_perf_stat_inc(&perf_stats.debug_sqbs_incomplete);
                goto again;
        }
        if (rc < 0) {
                DBF_ERROR("%4x SQBS ERROR", SCH_NO(q));
                DBF_ERROR("%3d%3d%2d", count, tmp_count, nr);
                q->handler(q->irq_ptr->cdev,
                           QDIO_ERROR_ACTIVATE_CHECK_CONDITION,
                           0, -1, -1, q->irq_ptr->int_parm);
                return 0;
        }
        WARN_ON(tmp_count);
        return count - tmp_count;
}
```

qdio_do_sqbs() is used in function

SQBS is used in the generic function qdio_main.c/set_buf_state. set_buf_state will call set_buf_states which will determine if the program can use SQBS or must set the state itself. Where set_buf_state or set_buf_states are used implies the situations in which SQBS is used.

set_buf_state references:

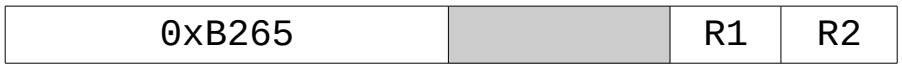
qdio_main.c/qdio_stop_polling
qdio_main.c/inbound_primed

set_buf_states references:

qdio_main.c/set_buf_state
qdio_main.c/init_buf_states
qdio_main.c/qdio_stop_polling
qdio_main.c/inbound_primed
qdio_main.c/handle_inbound
qdio_main.c/handle_outbound

SVS – SUBSYSTEM VARY STATUS

```
SVS R1,R2
```



SUBSYSTEM VARY STATUS is provided with TDD. Adapter interrupt facility indicators are manipulated by SVS. Operands 1 and 2 are even odd register pairs. Operand 1 identifies the function and targeted indicator and Operand 2 provided information about the state of the indicator.

Most of the description below is based upon assumptions and guesses of how this instruction might operate and may be inaccurate in the general case. Most of this information is inferred from the in-line assembler template in the example.

Operand 1 – Even register

Identifies the targeted indicator.

This is a guess based upon the apparent context of this instruction. How such an identification is supplied by the program is unclear.

Operand 1 – Odd register

A function code is supplied by Operand 1's odd registers

Operand 2 – Even register

The value of the indicator (before or after instruction execution is unclear and may be dependent upon the function)

Operand 2 – Odd register

A time value is stored.

Whether this represents the time since the last interrupt or the time until the next

scheduled interrupt or something else is possible.

SVS Clear Global Summary (Function 3)

Clear Global Summary clears any pending adapter interrupts implied by accumulated triggering of the summary indicator. Operand 1's even register contents are ignored.

Example usage:

```
static inline unsigned long do_clear_global_summary(void)
{
      register unsigned long __fn asm("1") = 3;
      register unsigned long __tmp asm("2");
      register unsigned long __time asm("3");

      asm volatile(
            "     .insn rre,0xb2650000,2,0"
            : "+d" (__fn), "=d" (__tmp), "=d" (__time));
      return __time;
}
```

Linux only uses the second operand's odd register when using SVS. Data returned by the instruction, indicated by the “+” or “-” constraints are ignored in the Linux example. Other functions this instruction might perform are purely speculative.

Reference:

drivers/s390/cio/qdio_thinint.c/tiqdio_thinint_handler

Architecture and Evolution

Although there were numerous networking enhancements between OSA and OSA Express, the major feature change was the incorporation of QDIO. The read/write subchannels remain in OSA Express adapters. Many more network configuration commands were added by OSA Express to allow dynamic configuration of the adapter (commands that might have been similar with OSA, but static and issued by a proprietary utility) for Layer-2 or Layer-3 operations. However, network frames and packets no longer use the read/write subchannels but utilize the new QDIO interface. Although a detailed comparison between OSA and OSA Express has not been done, it is likely that the adapter network commands and responses are very similar in overall structure and design to the original OSA, non-QDIO adapters. It is abundantly obvious that OSA Express adapters were not a wholesale functional redesign, at least as far the the program is concerned, but merely added QDIO to the OSA adapters. The functional interface of OSA adapters was the foundation for OSA Express.

This section provides some speculation on the internal implementation and evolution of the QDIO program interface. This section also approaches the topic of potential design decisions that will need to be made with regard to potential emulation of QDIO and network adapters that utilize QDIO. To some degree these discussions also expose the limitations that exist with source code reverse engineering.

Adapter Main Storage Interface

Examination of the Linux zfcp and qeth implementations suggests the data device is emulated by the channel subsystem. Both implementations use the single Linux qdio driver for the data device. In hardware terms, if the SCSI fiber channel adapter and the OSA Express adapters actually implemented the identical interface themselves, then each would have to have common code to do so. While this is entirely possible, it is also easy to conceive that the QDIO interface itself is external to the adapters and the adapters themselves utilize a different proprietary interface internal to the mainframe for communication with the channel subsystem.

The channel subsystem is in fact a program that utilizes the CPU on a real mainframe and the adapters are part of the mainframe I/O cage. Where the exact functional boundary exists is not significant from the point of view of program usage. However, it is informative with regards to a potential emulation of the feature.

Historically, the OSA Express adapter was the first to utilize QDIO. OSA Express devices were first available on ESA/390 systems. Later, fiber channel SCSI devices were added under z/Architecture. There is one strikingly unique difference between the SCSI and OSA Express usages of QDIO. The System Buffer Address List (SBAL) is strictly read-only when QDIO is used by OSA Express devices. However, the SCSI usage of QDIO actually modifies data in the System Buffer Address List as does Hipersocket adapters. This suggests that one of the innovations that occurred internal to the mainframe was providing access by the adapter to the SBAL, either directly by providing its address or indirectly by providing some

other mechanism that allows the SBAL entry content to be accessed and altered by the adapter.

One of the major enhancements of OSA Express 3 over OSA Express 2 is the data router feature. This OSA feature allows the adapter to store network frames or packets directly into main storage. Previously this was not possible. This suggests that originally the interface between the adapter and the channel subsystem did not expose the SBAL to the adapter, and hence, the QDIO feature in the channel subsystem had act as the intermediary between the program's usage of main storage for QDIO operations and the physical hardware adapter, receiving data from the earlier OSA Express adapters and writing the date to main storage or reading the data from main storage and providing it to the adapter.

This perspective is consistent with published descriptions of the data router feature and the differences between how OSA Express 2 and 3 operate. While IBM has articulated that OSA Express 2 queues the network packet or frame while OSA Express 3 does not, the reality is that the adapter must queue the Layer-2 frame read from the external network and then handle it as a Layer-2 frame or a Layer-3 packet. The implementation reality is more likely that the real queuing in the earlier OSA Express adapters was the result of the QDIO-to-adapter interface. Clearly the adapter had the ability to deliver to physical main storage data, because the Channel Subsystem operates from physical main storage, just different portions of it than does a program executing in a logical partition that interacts with a QDIO adapter.

This second observation is also helpful in defining an emulation of QDIO and how it might be structured. It also is consistent with the view that QDIO exists as a separate implementation from the the adapter itself as suggested above.

QEBSM Considerations

It is difficult looking at the Linux QDIO driver what capabilities are provided strictly in a z/VM environment and those available in a LPAR configuration. The following has been identified as elements that exist in the z/VM environment:

1. SIE assists for QDIO, likely enabled is the SIE control block. Note a new SIE option in the has been identified in use by Linux KVM. It is possible this enables QDIO assists for either the KVM case in an LPAR or the vSIE z/VM case when Linux KVM is executing in a z/VM virtual machine.
2. QEBSM CPU instruction (requires SIE QDIO assists to be enabled)
3. Millicode execution of SIGNAL ADAPTER Input or Output functions or elimination entirely of the need to issue SIGNAL ADAPTER instructions when QEBSM is being used. Elimination of SIGNAL ADAPTER instruction use is possible via the adapter characteristics supplied by the channel subsystem. Some scenarios must exist that eliminate some of all of the SIGNAL adapter uses because the Linux QDIO driver code supports this. The z/VM descriptions do not make it clear which is the case. The descriptions **do** make it clear that z/VM is not entered for virtual machine QDIO transfers when QEBSM is used with QDIO assists.

   Further, it is clear where the SIGNAL ADAPTER Output function is involved, a case

exists where the QEBSM token is used. In this case SIGA has not been eliminated. I suspect that the real differentiator for QDIO assistance with SIGA is whether the subchannel token is used or the subchannel ID. In the case of the former, QDIO Assistance is provided.

4. Automatic SIGNAL ADAPTER Synchronize when an Adapter Interrupt occurs
5. Automatic SIGNAL ADAPTER Synchronize when an PCI output interrupt occurs

All of these capabilities are focused on performance improvements, not functional improvements. The last two options could be provided by z/VM (or PR/SM) without the need of SIE assists. If so, it is also clear that z/VM can emulate Adapter Interrupts. The use of the term “hypervisor” in the Linux comments also makes it unclear whether z/VM is being referenced or PR/SM, the LPAR case. Note PR/SM performance enhancements would also benefit z/VM that could utilize them as well as Linux running in an LPAR.

For QEBSM to be effective, it requires that QDIO Assists are enabled by z/VM for the virtual machine. This suggests that QEBSM is not part of an LPAR implementation. The following table attempts to articulate the potential options:

Performance Enhancement	LPAR	z/VM Virtual Machine
QDIO SIE Assists	Not applicable	Applicable
QEBSM	Not applicable	Applicable
SIGA SIE support	Not applicable	Applicable
QEBSM elimination of SIGA	Not applicable	Possible
Adapter Interrupt Facility	Applicable	Applicable
Sync on Adapter Interrupt	Possible	Possible
Sync on PCI out Interrupt	Possible	Possible

The entire question of z/VM Guest LAN or vSWITCH emulation and the underlying role of SIE and z/VM's support of it raises a number of unanswerable questions concerning how this functions. The SIE question is really only a concern from the perspective of enabling QDIO Assists within an emulation to support execution of z/VM in the emulated environment. While much effort has been applied to determining how QEBSM operates in this document, it is clear that for an LPAR emulation of QDIO, it is not required and maybe is not even available. Many unanswered questions remain for an emulation of SIE QDIO Assists for z/VM in addition to the QEBSM instruction itself.

Without QIOAssists support, z/VM would operate with its traditional pre-QEBSM functionality.

The subchannel token would appear to provide the key linkage between the z/VM virtual machine, its virtual OSA Express adapter, its shadow tables and network operations with the physical installed OSA Express adapter. How exactly the token is established and to what it actually relates is impossible to ascertain from the Linux source code.

Adapter Functionality

All adapter functionality is defined by the

SENSE ID information that defines the adapter (OSA Express vs. Hipersocket vs. OSN interface),

READ CONFIGURATION DATA configuration record (physical vs. z/VM emulation),

set of adapter characteristics and

general channel subsystem characteristics (QDIO, AIF, TDD).

An adapter emulation can control how the program interacts with the emulated adapter by controlling what is presented through these four sources of program information.

Similar approaches are possible for network configuration not addressed in detail in this document.

Appendix A – GCC Useful References

This section provides an overview of key GCC facilities and their relationship to understanding the operation of the Linux QETH Driver.

s390 32-bit ABI Types

ABI Type	sizeof() in bytes	Alignment	zSeries Type	GNU Assembler
char	1	1	byte	.byte
short	2	2	halfword	.hword
int	4	4	(full)word	.long
long	4	4	(full)word	.long
long long	8	8	doubleword	.quad
pointer	4	4	(full)word	.long

s390 64-bit ABI Types

ABI Type	sizeof() in bytes	Alignment	zSeries Type	GNU Assembler
char	1	1	byte	.byte
short	2	2	halfword	.hword
int	4	4	(full)word	.long
long	8	8	doubleword	.quad
pointer	8	8	doubleword	.quad

In-line Assembler

In-line assembler is defined by the asm statement. “Volatile” will ensure the compiler does not eliminate the code.

```
asm [volatile]
    (assembler-template
      : output-constraint [,output-constraint]
      : input-constraint [,input-constraint]
      : clobber [,clobber]
     )
```

Explicit register variables are established with a declaration:

```
register long variable asm (“n”);
```

The above statement will assign “variable” to the machine register n.

To place a value in the register:

```
register long variable asm (“n”) = parameter;
```

Constraints are applied to the assembler-template instruction(s) by identifying the input or output constraint that applies. Constraints are numbered 0 to n starting with the first output constraint as 0 and continuing to the end of the constraints.

Constraints

Constraints are specified by a constraint string followed by the a C expression in parenthesis:

```
“x” (variable)
```

Constraint	Type	Meaning
m	simple	Memory operand allowed
o	simple	An offsetable memory operand is allowed
V	simple	A memory operand that is not offsetable
<	simple	A memory operand with autodecrement addressing
>	simple	A memory operand with autoincrement addressing
r	simple	A register operand that is a general register
i	simple	An immediate operand is allowed.
n	simple	An immediate operand with a known value is allowed.
s	simple	An immediate operand that is not an explicit integer. May be accompanied by additional constraints.
g	simple	Any register, memory or immediate integer is allowed, except for registers that are not general registers.
X	simple	Any operand whatsoever is allowed.
p	simple	An operand that is a valid memory address if allowed.
=	modifier	Operand is write-only.
+	modifier	Operand is both read and write.

Constraint	Type	Meaning
a	machine	Address register (general purpose register other than 0)
c	machine	Condition code register.
d	machine	Data register
f	machine	Floating point register
I	machine	Unsigned 8-bit constant (0-255)
J	machine	Unsigned 12-bit constant (0-4095)
K	machine	Signed 16-bit constant (-32768-32767)
L	machine	Value appropriate for displacement, short (0..4095) or long (-524288..524287)
M	machine	Constant integer with a value of 0x7fffffff
N[0-9][H,Q] [D,S,H][0,F]	machine	Multiple part constraint: 0-9 – Number of the part counting from the least significant H,Q – Mode of the part D,S,H – Mode of the containing operand 0,F – value of the other parts (F=all bits set) The constraint matches if the specified part of a constant has a value different from its other parts.
Q	machine	Memory reference without index register and with short displacement
R	machine	Memory reference with index register and short displacement
S	machine	Memory reference without index register and with long displacement
T	machine	Memory reference with index register and with long displacement
U	machine	Pointer with short displacement
W	machine	Pointer with long displacement
Y	machine	Shift count operand

Appendix B – Linux Modules

QDIO support is provided by three modules in the directory drivers/s390/cio:

qdio_main.c – Support for QDIO operations

qdio_setup.c – QDIO device initialization functions used by qdio_main.c

qdio_thinint.c – Generic QDIO Adapter Interrupt Facility interrupt handler

Other driver modules only communicate with qdio_main.c directly. The following table shows the relationship between qdio_main.c and the adapter specific driver modules.

LIne	qdio_main.c	qeth_core_main.c	zfcp_qdio.c
265	qdio_init_buf_states		
	qdio_establish		
979	**qdio_get_ssqd_desc**		
		qeth_core_hardsetup_card	
1000	**qdio_cleanup**		
		qeth_qdio_clear_card	
1035	**qdio_shutdown**		
			zfcp_qdio_close
			zfcp_qdio_open
1109	**qdio_free**		
	qdio_cleanup		zfcp_qdio_destroy
	qdio_initialize		
1138	**qdio_initialize**		
		qeth_qdio_establish	
1157	**qdio_allocate**		
	qdio_initialize		
1217	**qdio_establish**		
	qdio_initialize	qeth_mpc_initialize	zfcp_qdio_open
		qeth_qdio_establish	
1293	**qdio_activate**		
		qeth_qdio_activate	zfcp_qdio_open
1505	**do_QDIO**		
		qeth_init_qdio_queues	zfcp_qdio_resp_put_back
		qeth_flush_buffers	zfcp_qdio_send
			zfcp_qdio_open

Appendix C – Linux Interrupt and I/O Handling

QETH Handlers

QETH devices use two forms of handlers, queue handlers and I/O interrupt handlers.

Queue Handlers

Which discipline (Layer-2 or Layer-3) an adapter will be configured to use will dictate the input and output discipline handler. The discipline handlers provide queue processing for input or output.

Layer-2:

qeth_l2_main.c/qeth_l2_qdio_input_handler,

qeth_core_main.c/qeth_qdio_output_handler

Layer-3:

qeth_l3_main.c/qeth_l3_qdio_input_handler,

qeth_core_main.c/qeth_qdio_output_handler

The QDIO driver uses a standard structure to initialize the QDIO operations. The QETH handler's input_handler and output_handler are supplied in the initialization data as specified in the card's discipline structure.

The QDIO driver manages a structure for each queue. This queue structure contains a handler field that points to the card's discipline input handler for input queues or the card's discipline output handler for output queues. This handler is usually referenced by q->handler.

I/O Interrupt Handlers

The CCW device also has a handler to process device hardware I/O interrupts:

qeth_core_main.c/qeth_irq (QETH read/write devices)

qdio_main.c/qdio_init_handler (when QDIO operations are initialized)

The second QDIO interrupt handler replaces the default qeth_irq handler used by the QETH driver prior to the adapter's data device has been initialized.

I/O interrupts in general also have a handler:

cio.c/do_IRQ

The general I/O interrupt handler will call either the registered Adapter Interrupt handler or the CCW device handler.

Adapter Interrupt Handlers

If Adapter Interrupts are used by the adapter, the QDIO driver will establish and register an adapter interrupt that handles all QDIO adapter interrupts. Linux bases this on the I/O subclass. QDIO adapter interrupts are handled by:

qdio_thinint.c/tiqdio_thinint_handler

I/O Interrupts

All input/output interrupts are handled by drivers/s390/cio/cio.c's do_IRQ function.

When an adapter interrupt is identified it is passed to the do_adapter_IO function of drivers/s390/cio/airq.c. Adapter interrupt handling starts with the do_IRQ function.

Standard I/O interrupts are passed to the driver interrupt handler registered for the subchannel, sch->driver->irq(sch). The address of subchannel structure is located from the interrupt parameter. Linux uses the address of the subchannel structure as the interrupt parameter.

QDIO Adapter Interrupts

Adapter interrupts are handled by drivers/s390/cio/airq.c's do_adapter_IO function. A state change for any one of the adapter's queue is indicated by a one byte counter being incremented. An indicator's location is set by means of the Set Subchannel Indicator CHSC call issued to the adapter's data device. Each adapter interrupt has its own handler when registered. The handler is associated in Linux to the I/O subclass associated with the adapter interrupt. QDIO uses drivers/s390/cio/qdio_thinint.c/tiqdio_thinint_handler to process adapter interrupts.

QDIO Data Device I/0 Interrupt Handlers

Type	Input Processing	Output Processing
I/O interrupt	cio/cio.c/do_IRQ	cio/cio.c/do_IRQ
I/O interrupt handler	cio/qdio_main.c/qdio_int_handler	net/qdio_main.c/qdio_int_handler
PCI interrupt handler	cio/qdio_main.c/qdio_int_handler_pci	cio/qdio_main.c/qdio_int_handler_pci
I/O interrupt tasklet	cio/qdio_main.c/qdio_inbound_processing	cio/qdio_main.c/qdio_outbound_processing

The I/O interrupt tasklet will call the appropriate adapter queue handler.

QDIO Data Device Adapter Interrupt Handlers

Type	Input Processing	Output Processing
Adapter Interrupt	airq.c/do_adapter_IO	Not supported for outbound
QDIO adapter ints.	cio/qdio_thinint.c/tiqdio_thinint_handler	Not supported for outbound

Type	Input Processing	Output Processing
Adapter Int. tasklet	cio/qdio_main.c/tiqdio_inbound_processing	Not supported for outbound

The Adapter interrupt tasklet will call the appropriate adapter queue handler.

QETH Adapter Queue Handlers

QETH queue handlers are called by either the I/O interrupt tasklet or the Adapter interrupt tasklet.

Type	Input Processing	Output Processing
Layer-2 Queues	net/qeth_l2_main.c/ qeth_l2_qdio_input_handler	net/qeth_core_main.c/ qeth_qdio_output_handler
Layer-2 Buffer	net/qeth_l2_main.c/ qeth_l2_process_inbound_buffer	
Layer-3 Queues	net/qeth_l3_main.c/ qeth_l3_qdio_input_handler	net/qeth_core_main.c/ qeth_qdio_output_handler

QETH Read/Write Device I/O Interrupt Handlers

The QETH driver provides its own interrupt handler for the pair of channel interface devices used to configure the network adapters.

Type	Input Processing	Output Processing
I/O interrupt	cio/cio.c/do_IRQ	cio/cio.c/do_IRQ
Read/Write Device	net/qeth_core_main.c/qeth_irq	net/qeth_core_main.c/qeth_irq