Linux S390 Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

LINUX FOR S/390 AND ZSERIES

Common Device Support (CDS) Device Driver I/O Support Routines

Authors:

- Ingo Adlung
- Cornelia Huck

1.1 Introduction

This document describes the common device support routines for Linux/390. Different than other hardware architectures, ESA/390 has defined a unified I/O access method. This gives relief to the device drivers as they don't have to deal with different bus types, polling versus interrupt processing, shared versus non-shared interrupt processing, DMA versus port I/O (PIO), and other hardware features more. However, this implies that either every single device driver needs to implement the hardware I/O attachment functionality itself, or the operating system provides for a unified method to access the hardware, providing all the functionality that every single device driver would have to provide itself.

The document does not intend to explain the ESA/390 hardware architecture in every detail.This information can be obtained from the ESA/390 Principles of Operation manual (IBM Form. No. SA22-7201).

In order to build common device support for ESA/390 I/O interfaces, a functional layer was introduced that provides generic I/O access methods to the hardware.

The common device support layer comprises the I/O support routines defined below. Some of them implement common Linux device driver interfaces, while some of them are ESA/390 platform specific.

Note: In order to write a driver for S/390, you also need to look into the interface described in Documentation/s390/driver-model.rst.

Note for porting drivers from 2.4:

The major changes are:

- The functions use a ccw_device instead of an irq (subchannel).
- All drivers must define a ccw_driver (see driver-model.txt) and the associated functions.

- request_irq() and free_irq() are no longer done by the driver.
- The oper_handler is (kindof) replaced by the probe() and set_online() functions of the ccw_driver.
- The not_oper_handler is (kindof) replaced by the remove() and set_offline() functions of the ccw_driver.
- The channel device layer is gone.
- The interrupt handlers must be adapted to use a ccw_device as argument. Moreover, they don't return a devstat, but an irb.
- Before initiating an io, the options must be set via ccw_device_set_options().
- Instead of calling read_dev_chars()/read_conf_data(), the driver issues the channel program and handles the interrupt itself.

ccw_device_get_ciw() get commands from extended sense data.

ccw_device_start(), ccw_device_start_timeout(), ccw_device_start_key(), ccw_device_sta initiate an I/O request.

ccw_device_resume() resume channel program execution.

ccw_device_halt() terminate the current I/O request processed on the device.

do_IRQ() generic interrupt routine. This function is called by the interrupt entry routine whenever an I/O interrupt is presented to the system. The do_IRQ() routine determines the interrupt status and calls the device specific interrupt handler according to the rules (flags) defined during I/O request initiation with do_IO().

The next chapters describe the functions other than do_IRQ() in more details. The do_IRQ() interface is not described, as it is called from the Linux/390 first level interrupt handler only and does not comprise a device driver callable interface. Instead, the functional description of do_IO() also describes the input to the device specific interrupt handler.

Note: All explanations apply also to the 64 bit architecture s390x.

1.2 Common Device Support (CDS) for Linux/390 Device Drivers

1.2.1 General Information

The following chapters describe the I/O related interface routines the Linux/390 common device support (CDS) provides to allow for device specific driver implementations on the IBM ESA/390 hardware platform. Those interfaces intend to provide the functionality required by every device driver implementation to allow to drive a specific hardware device on the ESA/390 platform. Some of the interface routines are specific to Linux/390 and some of them can be found on other Linux platforms implementations too. Miscellaneous function prototypes, data declarations, and macro definitions can be found in the architecture specific C header file linux/arch/s390/include/asm/irq.h.

1.2.2 Overview of CDS interface concepts

Different to other hardware platforms, the ESA/390 architecture doesn't define interrupt lines managed by a specific interrupt controller and bus systems that may or may not allow for shared interrupts, DMA processing, etc.. Instead, the ESA/390 architecture has implemented a so called channel subsystem, that provides a unified view of the devices physically attached to the systems. Though the ESA/390 hardware platform knows about a huge variety of different peripheral attachments like disk devices (aka. DASDs), tapes, communication controllers, etc. they can all be accessed by a well defined access method and they are presenting I/O completion a unified way : I/O interruptions. Every single device is uniquely identified to the system by a so called subchannel, where the ESA/390 architecture allows for 64k devices be attached.

Linux, however, was first built on the Intel PC architecture, with its two cascaded 8259 programmable interrupt controllers (PICs), that allow for a maximum of 15 different interrupt lines. All devices attached to such a system share those 15 interrupt levels. Devices attached to the ISA bus system must not share interrupt levels (aka. IRQs), as the ISA bus bases on edge triggered interrupts. MCA, EISA, PCI and other bus systems base on level triggered interrupts, and therewith allow for shared IRQs. However, if multiple devices present their hardware status by the same (shared) IRQ, the operating system has to call every single device driver registered on this IRQ in order to determine the device driver owning the device that raised the interrupt.

Up to kernel 2.4, Linux/390 used to provide interfaces via the IRQ (subchannel). For internal use of the common I/O layer, these are still there. However, device drivers should use the new calling interface via the ccw_device only.

During its startup the Linux/390 system checks for peripheral devices. Each of those devices is uniquely defined by a so called subchannel by the ESA/390 channel subsystem. While the subchannel numbers are system generated, each subchannel also takes a user defined attribute, the so called device number. Both subchannel number and device number cannot exceed 65535. During sysfs initialisation, the information about control unit type and device types that imply specific I/O commands (channel command words - CCWs) in order to operate the device are gathered. Device drivers can retrieve this set of hardware information during their initialization step to recognize the devices they support using the information saved in the struct ccw_device given to them. This methods implies that Linux/390 doesn't require to probe for free (not armed) interrupt request lines (IRQs) to drive its devices with. Where applicable, the device drivers can use issue the READ DEVICE CHARACTERISTICS ccw to retrieve device characteristics in its online routine.

In order to allow for easy I/O initiation the CDS layer provides a ccw_device_start() interface that takes a device specific channel program (one or more CCWs) as input sets up the required architecture specific control blocks and initiates an I/O request on behalf of the device driver. The ccw_device_start() routine allows to specify whether it expects the CDS layer to notify the device driver for every interrupt it observes, or with final status only. See ccw_device_start() for more details. A device driver must never issue ESA/390 I/O commands itself, but must use the Linux/390 CDS interfaces instead.

For long running I/O request to be canceled, the CDS layer provides the

ccw_device_halt() function. Some devices require to initially issue a HALT SUBCHANNEL (HSCH) command without having pending I/O requests. This function is also covered by ccw_device_halt().

get_ciw() - get command information word

This call enables a device driver to get information about supported commands from the extended SenseID data.

```
struct ciw *
ccw_device_get_ciw(struct ccw_device *cdev, __u32 cmd);
```

cdev	The ccw_device for which the command is to be retrieved.
cmd	The command type to be retrieved.

ccw_device_get_ciw() returns:

NULL	No extended data available, invalid device or command not found.
!NULL	The command requested.

```
ccw_device_start() - Initiate I/O Request
```

The ccw_device_start() routines is the I/O request front-end processor. All device driver I/O requests must be issued using this routine. A device driver must not issue ESA/390 I/O commands itself. Instead the ccw_device_start() routine provides all interfaces required to drive arbitrary devices.

This description also covers the status information passed to the device driver's interrupt handler as this is related to the rules (flags) defined with the associated I/O request when calling ccw_device_start().

```
int ccw_device_start(struct ccw_device *cdev,
                     struct ccw1 *cpa,
                     unsigned long intparm,
                     __u8 lpm,
                     unsigned long flags);
int ccw_device_start_timeout(struct ccw_device *cdev,
                             struct ccw1 *cpa,
                             unsigned long intparm,
                             __u8 lpm,
                             unsigned long flags,
                             int expires);
int ccw_device_start_key(struct ccw_device *cdev,
                         struct ccw1 *cpa,
                         unsigned long intparm,
                         __u8 lpm,
                         __u8 key,
                         unsigned long flags);
int ccw_device_start_key_timeout(struct ccw_device *cdev,
                                 struct ccw1 *cpa,
                                 unsigned long intparm,
                                 __u8 lpm,
                                 __u8 key,
```

(continues on next page)

(continued from previous page)

```
                                    unsigned long flags,
                                    int expires);
```

cdev	ccw_device the I/O is destined for
cpa	logical start address of channel program
user_intparm	user specific interrupt information; will be presented back to the device driver's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.
lpm	defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.
key	the storage key to use for the I/O (useful for operating on a storage with a storage key != default key)
flag	defines the action to be performed for I/O processing
expires	timeout value in jiffies. The common I/O layer will terminate the running program after this and call the interrupt handler with ERR_PTR(-ETIMEDOUT) as irb.

Possible flag values are:

DOIO_ALLOW_SUSPEND	the channel program may become suspended
DOIO_DENY_PREFETCH	don't allow for CCW prefetch; usually this implies the channel program might become modified
DOIO_SUPPRESS_INTER	don't call the handler on intermediate status

The cpa parameter points to the first format 1 CCW of a channel program:

```
struct ccw1 {
      __u8  cmd_code;/* command code */
      __u8  flags;   /* flags, like IDA addressing, etc. */
      __u16 count;   /* byte count */
      __u32 cda;     /* data address */
} __attribute__ ((packed,aligned(8)));
```

with the following CCW flags values defined:

CCW_FLAG_DC	data chaining
CCW_FLAG_CC	command chaining
CCW_FLAG_SLI	suppress incorrect length
CCW_FLAG_SKIP	skip
CCW_FLAG_PCI	PCI
CCW_FLAG_IDA	indirect addressing
CCW_FLAG_SUSPEND	suspend

Via ccw_device_set_options(), the device driver may specify the following options for the device:

DOIO_EARLY_NOTIFICATION	allow for early interrupt notification
DOIO_REPORT_ALL	report all interrupt conditions

The ccw_device_start() function returns:

0	successful completion or request successfully initiated
-EBUSY	The device is currently processing a previous I/O request, or there is a status pending at the device.
-ENODEV	cdev is invalid, the device is not operational or the ccw_device is not online.

When the I/O request completes, the CDS first level interrupt handler will accumulate the status in a struct irb and then call the device interrupt handler. The intparm field will contain the value the device driver has associated with a particular I/O request. If a pending device status was recognized, intparm will be set to 0 (zero). This may happen during I/O initiation or delayed by an alert status notification. In any case this status is not related to the current (last) I/O request. In case of a delayed status notification no special interrupt will be presented to indicate I/O completion as the I/O request was never started, even though ccw_device_start() returned with successful completion.

The irb may contain an error value, and the device driver should check for this first:

-ETIMEDOUT	the common I/O layer terminated the request after the specified timeout value
-EIO	the common I/O layer terminated the request due to an error state

If the concurrent sense flag in the extended status word (esw) in the irb is set, the field erw.scnt in the esw describes the number of device specific sense bytes available in the extended control word irb->scsw.ecw[]. No device sensing by the device driver itself is required.

The device interrupt handler can use the following definitions to investigate the primary unit check source coded in sense byte 0 :

SNS0_CMD_REJECT	0x80
SNS0_INTERVENTION_REQ	0x40
SNS0_BUS_OUT_CHECK	0x20
SNS0_EQUIPMENT_CHECK	0x10
SNS0_DATA_CHECK	0x08
SNS0_OVERRUN	0x04
SNS0_INCOMPL_DOMAIN	0x01

Depending on the device status, multiple of those values may be set together. Please refer to the device specific documentation for details.

The irb->scsw.cstat field provides the (accumulated) subchannel status :

SCHN_STAT_PCI	program controlled interrupt
SCHN_STAT_INCORR_LEN	incorrect length
SCHN_STAT_PROG_CHECK	program check
SCHN_STAT_PROT_CHECK	protection check
SCHN_STAT_CHN_DATA_CHK	channel data check
SCHN_STAT_CHN_CTRL_CHK	channel control check
SCHN_STAT_INTF_CTRL_CHK	interface control check
SCHN_STAT_CHAIN_CHECK	chaining check

The irb->scsw.dstat field provides the (accumulated) device status :

DEV_STAT_ATTENTION	attention
DEV_STAT_STAT_MOD	status modifier
DEV_STAT_CU_END	control unit end
DEV_STAT_BUSY	busy
DEV_STAT_CHN_END	channel end
DEV_STAT_DEV_END	device end
DEV_STAT_UNIT_CHECK	unit check
DEV_STAT_UNIT_EXCEP	unit exception

Please see the ESA/390 Principles of Operation manual for details on the individual flag meanings.

Usage Notes:

ccw_device_start() must be called disabled and with the ccw device lock held.

The device driver is allowed to issue the next ccw_device_start() call from within its interrupt handler already. It is not required to schedule a bottom-half, unless a non deterministically long running error recovery procedure or similar needs to be scheduled. During I/O processing the Linux/390 generic I/O device driver support has already obtained the IRQ lock, i.e. the handler must not try to obtain it again when calling ccw_device_start() or we end in a deadlock situation!

If a device driver relies on an I/O request to be completed prior to start the next it can reduce I/O processing overhead by chaining a NoOp I/O command CCW_CMD_NOOP to the end of the submitted CCW chain. This will force Channel-End and Device-End status to be presented together, with a single interrupt. However, this should be used with care as it implies the channel will remain busy, not being able to process I/O requests for other devices on the same channel. Therefore e.g. read commands should never use this technique, as the result will be presented by a single interrupt anyway.

In order to minimize I/O overhead, a device driver should use the DOIO_REPORT_ALL only if the device can report intermediate interrupt information prior to device-end the device driver urgently relies on. In this case all I/O interruptions are presented to the device driver until final status is recognized.

If a device is able to recover from asynchronously presented I/O errors, it can perform overlapping I/O using the DOIO_EARLY_NOTIFICATION flag. While some devices always report channel-end and device-end together, with a single interrupt, others present primary status (channel-end) when the channel is ready for

the next I/O request and secondary status (device-end) when the data transmission has been completed at the device.

Above flag allows to exploit this feature, e.g. for communication devices that can handle lost data on the network to allow for enhanced I/O processing.

Unless the channel subsystem at any time presents a secondary status interrupt, exploiting this feature will cause only primary status interrupts to be presented to the device driver while overlapping I/O is performed. When a secondary status without error (alert status) is presented, this indicates successful completion for all overlapping ccw_device_start() requests that have been issued since the last secondary (final) status.

Channel programs that intend to set the suspend flag on a channel command word (CCW) must start the I/O operation with the DOIO_ALLOW_SUSPEND option or the suspend flag will cause a channel program check. At the time the channel program becomes suspended an intermediate interrupt will be generated by the channel subsystem.

ccw_device_resume() - Resume Channel Program Execution

If a device driver chooses to suspend the current channel program execution by setting the CCW suspend flag on a particular CCW, the channel program execution is suspended. In order to resume channel program execution the CIO layer provides the ccw_device_resume() routine.

```
int ccw_device_resume(struct ccw_device *cdev);
```

cdev	ccw_device the resume operation is requested for

The ccw_device_resume() function returns:

0	suspended channel program is resumed
-EBUSY	status pending
-ENODEV	cdev invalid or not-operational subchannel
-EINVAL	resume function not applicable
-ENOTCONN	there is no I/O request pending for completion

Usage Notes:

Please have a look at the ccw_device_start() usage notes for more details on suspended channel programs.

ccw_device_halt() - Halt I/O Request Processing

Sometimes a device driver might need a possibility to stop the processing of a long-running channel program or the device might require to initially issue a halt subchannel (HSCH) I/O command. For those purposes the ccw_device_halt() command is provided.

ccw_device_halt() must be called disabled and with the ccw device lock held.

```
int ccw_device_halt(struct ccw_device *cdev,
                    unsigned long intparm);
```

cdev	ccw_device the halt operation is requested for
intparm	interruption parameter; value is only used if no I/O is outstanding, otherwise the intparm associated with the I/O request is returned

The ccw_device_halt() function returns:

0	request successfully initiated
-EBUSY	the device is currently busy, or status pending.
-ENODEV	cdev invalid.
-EINVAL	The device is not operational or the ccw device is not online.

Usage Notes:

A device driver may write a never-ending channel program by writing a channel program that at its end loops back to its beginning by means of a transfer in channel (TIC) command (CCW_CMD_TIC). Usually this is performed by network device drivers by setting the PCI CCW flag (CCW_FLAG_PCI). Once this CCW is executed a program controlled interrupt (PCI) is generated. The device driver can then perform an appropriate action. Prior to interrupt of an outstanding read to a network device (with or without PCI flag) a ccw_device_halt() is required to end the pending operation.

```
ccw_device_clear() - Terminage I/O Request Processing
```

In order to terminate all I/O processing at the subchannel, the clear subchannel (CSCH) command is used. It can be issued via ccw_device_clear().

ccw_device_clear() must be called disabled and with the ccw device lock held.

```
int ccw_device_clear(struct ccw_device *cdev, unsigned long intparm);
```

cdev	ccw_device the clear operation is requested for
intparm	interruption parameter (see ccw_device_halt())

The ccw_device_clear() function returns:

0	request successfully initiated
-ENODEV	cdev invalid
-EINVAL	The device is not operational or the ccw device is not online.

1.2.3 Miscellaneous Support Routines

This chapter describes various routines to be used in a Linux/390 device driver programming environment.

get_ccwdev_lock()

Get the address of the device specific lock. This is then used in spin_lock() / spin_unlock() calls.

```
__u8 ccw_device_get_path_mask(struct ccw_device *cdev);
```

Get the mask of the path currently available for cdev.

CHAPTER

TWO

IBM 3270 DISPLAY SYSTEM SUPPORT

This file describes the driver that supports local channel attachment of IBM 3270 devices. It consists of three sections:

- Introduction
- Installation
- Operation

2.1 Introduction

This paper describes installing and operating 3270 devices under Linux/390. A 3270 device is a block-mode rows-and-columns terminal of which I'm sure hundreds of millions were sold by IBM and clonemakers twenty and thirty years ago.

You may have 3270s in-house and not know it. If you're using the VM-ESA operating system, define a 3270 to your virtual machine by using the command "DEF GRAF <hex-address>" This paper presumes you will be defining four 3270s with the CP/CMS commands:

- DEF GRAF 620
- DEF GRAF 621
- DEF GRAF 622
- DEF GRAF 623

Your network connection from VM-ESA allows you to use x3270, tn3270, or another 3270 emulator, started from an xterm window on your PC or workstation. With the DEF GRAF command, an application such as xterm, and this Linux-390 3270 driver, you have another way of talking to your Linux box.

This paper covers installation of the driver and operation of a dialed-in x3270.

2.2 Installation

You install the driver by installing a patch, doing a kernel build, and running the configuration script (config3270.sh, in this directory).

WARNING: If you are using 3270 console support, you must rerun the configuration script every time you change the console's address (perhaps by using the condev= parameter in silo's /boot/parmfile). More precisely, you should rerun the configuration script every time your set of 3270s, including the console 3270, changes subchannel identifier relative to one another. ReIPL as soon as possible after running the configuration script and the resulting /tmp/mkdev3270.

If you have chosen to make tub3270 a module, you add a line to a configuration file under /etc/modprobe.d/. If you are working on a VM virtual machine, you can use DEF GRAF to define virtual 3270 devices.

You may generate both 3270 and 3215 console support, or one or the other, or neither. If you generate both, the console type under VM is not changed. Use #CP Q TERM to see what the current console type is. Use #CP TERM CONMODE 3270 to change it to 3270. If you generate only 3270 console support, then the driver automatically converts your console at boot time to a 3270 if it is a 3215.

In brief, these are the steps:

1. Install the tub3270 patch
2. (If a module) add a line to a file in /etc/modprobe.d/*.conf
3. (If VM) define devices with DEF GRAF
4. Reboot
5. Configure

To test that everything works, assuming VM and x3270,

1. Bring up an x3270 window.
2. Use the DIAL command in that window.
3. You should immediately see a Linux login screen.

Here are the installation steps in detail:

1. The 3270 driver is a part of the official Linux kernel source. Build a tree with the kernel source and any necessary patches. Then do:

```
make oldconfig
(If you wish to disable 3215 console support, edit
.config; change CONFIG_TN3215's value to "n";
and rerun "make oldconfig".)
make image
make modules
make modules_install
```

2. (Perform this step only if you have configured tub3270 as a module.) Add a line to a file /etc/modprobe.d/*.conf to automatically load the driver when it's needed. With this line added, you will see login

prompts appear on your 3270s as soon as boot is complete (or with emulated 3270s, as soon as you dial into your vm guest using the command "DIAL <vmguestname>"). Since the line-mode major number is 227, the line to add should be:

```
alias char-major-227 tub3270
```

3. Define graphic devices to your vm guest machine, if you haven't already. Define them before you reboot (reipl):

- DEFINE GRAF 620
- DEFINE GRAF 621
- DEFINE GRAF 622
- DEFINE GRAF 623

4. Reboot. The reboot process scans hardware devices, including 3270s, and this enables the tub3270 driver once loaded to respond correctly to the configuration requests of the next step. If you have chosen 3270 console support, your console now behaves as a 3270, not a 3215.

5. Run the 3270 configuration script config3270. It is distributed in this same directory, Documentation/s390, as config3270.sh. Inspect the output script it produces, /tmp/mkdev3270, and then run that script. This will create the necessary character special device files and make the necessary changes to /etc/inittab.

Then notify /sbin/init that /etc/inittab has changed, by issuing the telinit command with the q operand:

```
cd Documentation/s390
sh config3270.sh
sh /tmp/mkdev3270
telinit q
```

This should be sufficient for your first time. If your 3270 configuration has changed and you're reusing config3270, you should follow these steps:

```
Change 3270 configuration
Reboot
Run config3270 and /tmp/mkdev3270
Reboot
```

Here are the testing steps in detail:

1. Bring up an x3270 window, or use an actual hardware 3278 or 3279, or use the 3270 emulator of your choice. You would be running the emulator on your PC or workstation. You would use the command, for example:

```
x3270 vm-esa-domain-name &
```

if you wanted a 3278 Model 4 with 43 rows of 80 columns, the default model number. The driver does not take advantage of extended attributes.

The screen you should now see contains a VM logo with input lines near the bottom. Use TAB to move to the bottom line, probably labeled "COMMAND ===>".

2. Use the DIAL command instead of the LOGIN command to connect to one of the virtual 3270s you defined with the DEF GRAF commands:

```
dial my-vm-guest-name
```

3. You should immediately see a login prompt from your Linux-390 operating system. If that does not happen, you would see instead the line "DIALED TO my-vm-guest-name 0620".

To troubleshoot: do these things.

A. Is the driver loaded? Use the lsmod command (no operands) to find out. Probably it isn't. Try loading it manually, with the command "insmod tub3270". Does that command give error messages? Ha! There's your problem.

B. Is the /etc/inittab file modified as in installation step 3 above? Use the grep command to find out; for instance, issue "grep 3270 /etc/inittab". Nothing found? There's your problem!

C. Are the device special files created, as in installation step 2 above? Use the ls -l command to find out; for instance, issue "ls -l /dev/3270/tty620". The output should start with the letter "c" meaning character device and should contain "227, 1" just to the left of the device name. No such file? no "c"? Wrong major number? Wrong minor number? There's your problem!

D. Do you get the message:

```
"HCPDIA047E my-vm-guest-name 0620 does not exist"?
```

If so, you must issue the command "DEF GRAF 620" from your VM 3215 console and then reboot the system.

2.3 OPERATION.

The driver defines three areas on the 3270 screen: the log area, the input area, and the status area.

The log area takes up all but the bottom two lines of the screen. The driver writes terminal output to it, starting at the top line and going down. When it fills, the status area changes from "Linux Running" to "Linux More···". After a scrolling timeout of (default) 5 sec, the screen clears and more output is written, from the top down.

The input area extends from the beginning of the second-to-last screen line to the start of the status area. You type commands in this area and hit ENTER to execute them.

The status area initializes to "Linux Running" to give you a warm fuzzy feeling. When the log area fills up and output awaits, it changes to "Linux More···". At this

time you can do several things or nothing. If you do nothing, the screen will clear in (default) 5 sec and more output will appear. You may hit ENTER with nothing typed in the input area to toggle between "Linux More···" and "Linux Holding", which indicates no scrolling will occur. (If you hit ENTER with "Linux Running" and nothing typed, the application receives a newline.)

You may change the scrolling timeout value. For example, the following command line:

```
echo scrolltime=60 > /proc/tty/driver/tty3270
```

changes the scrolling timeout value to 60 sec. Set scrolltime to 0 if you wish to prevent scrolling entirely.

Other things you may do when the log area fills up are: hit PA2 to clear the log area and write more output to it, or hit CLEAR to clear the log area and the input area and write more output to the log area.

Some of the Program Function (PF) and Program Attention (PA) keys are preassigned special functions. The ones that are not yield an alarm when pressed.

PA1 causes a SIGINT to the currently running application. You may do the same thing from the input area, by typing "^C" and hitting ENTER.

PA2 causes the log area to be cleared. If output awaits, it is then written to the log area.

PF3 causes an EOF to be received as input by the application. You may cause an EOF also by typing "^D" and hitting ENTER.

No PF key is preassigned to cause a job suspension, but you may cause a job suspension by typing "^Z"and hitting ENTER. You may wish to assign this function to a PF key. To make PF7 cause job suspension, execute the command:

```
echo pf7=^z > /proc/tty/driver/tty3270
```

If the input you type does not end with the two characters "^n", the driver appends a newline character and sends it to the tty driver; otherwise the driver strips the "^n" and does not append a newline. The IBM 3215 driver behaves similarly.

Pf10 causes the most recent command to be retrieved from the tube's command stack (default depth 20) and displayed in the input area. You may hit PF10 again for the next-most-recent command, and so on. A command is entered into the stack only when the input area is not made invisible (such as for password entry) and it is not identical to the current top entry. PF10 rotates backward through the command stack; PF11 rotates forward. You may assign the backward function to any PF key (or PA key, for that matter), say, PA3, with the command:

```
echo -e pa3=\\033k > /proc/tty/driver/tty3270
```

This assigns the string ESC-k to PA3. Similarly, the string ESC-j performs the forward function. (Rationale: In bash with vi-mode line editing, ESC-k and ESC-j retrieve backward and forward history. Suggestions welcome.)

Is a stack size of twenty commands not to your liking? Change it on the fly. To change to saving the last 100 commands, execute the command:

```
echo recallsize=100 > /proc/tty/driver/tty3270
```

Have a command you issue frequently? Assign it to a PF or PA key! Use the command:

```
echo pf24="mkdir foobar; cd foobar" > /proc/tty/driver/tty3270
```

to execute the commands mkdir foobar and cd foobar immediately when you hit PF24. Want to see the command line first, before you execute it? Use the -n option of the echo command:

```
echo -n pf24="mkdir foo; cd foo" > /proc/tty/driver/tty3270
```

Happy testing! I welcome any and all comments about this document, the driver, etc etc.

Dick Hitt <rbh00@utsglobal.com>

CHAPTER

THREE

S/390 DRIVER MODEL INTERFACES

3.1 1. CCW devices

All devices which can be addressed by means of ccws are called 'CCW devices' - even if they aren't actually driven by ccws.

All ccw devices are accessed via a subchannel, this is reflected in the structures under devices/:

```
devices/
   - system/
   - css0/
         - 0.0.0000/0.0.0815/
         - 0.0.0001/0.0.4711/
         - 0.0.0002/
         - 0.1.0000/0.1.1234/
         ...
         - defunct/
```

In this example, device 0815 is accessed via subchannel 0 in subchannel set 0, device 4711 via subchannel 1 in subchannel set 0, and subchannel 2 is a non-I/O subchannel. Device 1234 is accessed via subchannel 0 in subchannel set 1.

The subchannel named 'defunct' does not represent any real subchannel on the system; it is a pseudo subchannel where disconnected ccw devices are moved to if they are displaced by another ccw device becoming operational on their former subchannel. The ccw devices will be moved again to a proper subchannel if they become operational again on that subchannel.

You should address a ccw device via its bus id (e.g. 0.0.4711); the device can be found under bus/ccw/devices/.

All ccw devices export some data via sysfs.

cutype: The control unit type / model.

devtype: The device type / model, if applicable.

availability: Can be 'good' or 'boxed'; 'no path' or 'no device' for disconnected devices.

online: An interface to set the device online and offline. In the special case of the device being disconnected (see the notify function under 1.2), piping 0 to online will forcibly delete the device.

The device drivers can add entries to export per-device data and interfaces.

There is also some data exported on a per-subchannel basis (see under bus/css/devices/):

chpids: Via which chpids the device is connected.

pimpampom: The path installed, path available and path operational masks.

There also might be additional data, for example for block devices.

3.2 1.1 Bringing up a ccw device

This is done in several steps.

a. Each driver can provide one or more parameter interfaces where parameters can be specified. These interfaces are also in the driver's responsibility.

b. After a. has been performed, if necessary, the device is finally brought up via the 'online' interface.

3.3 1.2 Writing a driver for ccw devices

The basic struct ccw_device and struct ccw_driver data structures can be found under include/asm/ccwdev.h:

```
struct ccw_device {
      spinlock_t *ccwlock;
      struct ccw_device_private *private;
      struct ccw_device_id id;

      struct ccw_driver *drv;
      struct device dev;
      int online;

      void (*handler) (struct ccw_device *dev, unsigned long intparm,
                       struct irb *irb);
};

struct ccw_driver {
      struct module *owner;
      struct ccw_device_id *ids;
      int (*probe) (struct ccw_device *);
      int (*remove) (struct ccw_device *);
      int (*set_online) (struct ccw_device *);
      int (*set_offline) (struct ccw_device *);
      int (*notify) (struct ccw_device *, int);
      struct device_driver driver;
      char *name;
};
```

The 'private' field contains data needed for internal i/o operation only, and is not available to the device driver.

Each driver should declare in a MODULE_DEVICE_TABLE into which CU types/models and/or device types/models it is interested. This information can later be found in the struct ccw_device_id fields:

```
struct ccw_device_id {
      __u16   match_flags;

      __u16   cu_type;
      __u16   dev_type;
      __u8    cu_model;
      __u8    dev_model;

      unsigned long driver_info;
};
```

The functions in ccw_driver should be used in the following way:

probe: This function is called by the device layer for each device the driver is interested in. The driver should only allocate private structures to put in dev->driver_data and create attributes (if needed). Also, the interrupt handler (see below) should be set here.

```
int (*probe) (struct ccw_device *cdev);
```

Parameters:

cdev

- the device to be probed.

remove: This function is called by the device layer upon removal of the driver, the device or the module. The driver should perform cleanups here.

```
int (*remove) (struct ccw_device *cdev);
```

Parameters:

cdev

- the device to be removed.

set_online: This function is called by the common I/O layer when the device is activated via the 'online'attribute. The driver should finally setup and activate the device here.

```
int (*set_online) (struct ccw_device *);
```

Parameters:

cdev

- the device to be activated. The common layer has verified that the device is not already online.

set_offline: This function is called by the common I/O layer when the device is de-activated via the 'online' attribute. The driver should shut down the device, but not de-allocate its private data.

```
int (*set_offline) (struct ccw_device *);
```

Parameters:

cdev

- **the device to be deactivated. The common layer has** verified that the device is online.

notify: This function is called by the common I/O layer for some state changes of the device.

Signalled to the driver are:

- In online state, device detached (CIO_GONE) or last path gone (CIO_NO_PATH). The driver must return !0 to keep the device; for return code 0, the device will be deleted as usual (also when no notify function is registered). If the driver wants to keep the device, it is moved into disconnected state.
- In disconnected state, device operational again (CIO_OPER). The common I/O layer performs some sanity checks on device number and Device / CU to be reasonably sure if it is still the same device. If not, the old device is removed and a new one registered. By the return code of the notify function the device driver signals if it wants the device back: !0 for keeping, 0 to make the device being removed and re-registered.

```
int (*notify) (struct ccw_device *, int);
```

Parameters:

cdev

- the device whose state changed.

event

- the event that happened. This can be one of CIO_GONE, CIO_NO_PATH or CIO_OPER.

The handler field of the struct ccw_device is meant to be set to the interrupt handler for the device. In order to accommodate drivers which use several distinct handlers (e.g. multi subchannel devices), this is a member of ccw_device instead of ccw_driver. The handler is registered with the common layer during set_online() processing before the driver is called, and is deregistered during set_offline() after the driver has been called. Also, after registering / before deregistering, path grouping resp. disbanding of the path group (if applicable) are performed.

```
void (*handler) (struct ccw_device *dev, unsigned long intparm, struct irb␣
↪*irb);
```

Parameters: dev - the device the handler is called for

intparm - the intparm which allows the device driver to identify the i/o the interrupt is associated with, or to recognize the interrupt as unsolicited.

irb - interruption response block which contains the accumulated status.

The device driver is called from the common ccw_device layer and can retrieve information about the interrupt from the irb parameter.

3.4 1.3 ccwgroup devices

The ccwgroup mechanism is designed to handle devices consisting of multiple ccw devices, like lcs or ctc.

The ccw driver provides a 'group' attribute. Piping bus ids of ccw devices to this attributes creates a ccwgroup device consisting of these ccw devices (if possible). This ccwgroup device can be set online or offline just like a normal ccw device.

Each ccwgroup device also provides an 'ungroup' attribute to destroy the device again (only when offline). This is a generic ccwgroup mechanism (the driver does not need to implement anything beyond normal removal routines).

A ccw device which is a member of a ccwgroup device carries a pointer to the ccwgroup device in the driver_data of its device struct. This field must not be touched by the driver - it should use the ccwgroup device's driver_data for its private data.

To implement a ccwgroup driver, please refer to include/asm/ccwgroup.h. Keep in mind that most drivers will need to implement both a ccwgroup and a ccw driver.

3.5 2. Channel paths

Channel paths show up, like subchannels, under the channel subsystem root (css0) and are called 'chp0.<chpid>'. They have no driver and do not belong to any bus. Please note, that unlike /proc/chpids in 2.4, the channel path objects reflect only the logical state and not the physical state, since we cannot track the latter consistently due to lacking machine support (we don't need to be aware of it anyway).

status

- Can be 'online' or 'offline'. Piping 'on' or 'off' sets the chpid logically online/offline. Piping 'on' to an online chpid triggers path reprobing for all devices the chpid connects to. This can be used to force the kernel to re-use a channel path the user knows to be online, but the machine hasn't created a machine check for.

type

- The physical type of the channel path.

shared

- Whether the channel path is shared.

cmg

- The channel measurement group.

3.6 3. System devices

3.7 3.1 xpram

xpram shows up under devices/system/ as ‘xpram’.

3.8 3.2 cpus

For each cpu, a directory is created under devices/system/cpu/. Each cpu has an attribute ‘online’ which can be 0 or 1.

3.9 4. Other devices

3.10 4.1 Netiucv

The netiucv driver creates an attribute ‘connection’ under bus/iucv/drivers/netiucv. Piping to this attribute creates a new netiucv connection to the specified host.

Netiucv connections show up under devices/iucv/ as “netiucv<ifnum>”. The interface number is assigned sequentially to the connections defined via the ‘connection’ attribute.

user

- shows the connection partner.

buffer

- maximum buffer size. Pipe to it to change buffer size.

CHAPTER

FOUR

LINUX API FOR READ ACCESS TO Z/VM MONITOR RECORDS

Date : 2004-Nov-26

Author: Gerald Schaefer (geraldsc@de.ibm.com)

4.1 Description

This item delivers a new Linux API in the form of a misc char device that is usable from user space and allows read access to the z/VM Monitor Records collected by the *MONITOR System Service of z/VM.

4.2 User Requirements

The z/VM guest on which you want to access this API needs to be configured in order to allow IUCV connections to the *MONITOR service, i.e. it needs the IUCV *MONITOR statement in its user entry. If the monitor DCSS to be used is restricted (likely), you also need the NAMESAVE <DCSS NAME> statement. This item will use the IUCV device driver to access the z/VM services, so you need a kernel with IUCV support. You also need z/VM version 4.4 or 5.1.

There are two options for being able to load the monitor DCSS (examples assume that the monitor DCSS begins at 144 MB and ends at 152 MB). You can query the location of the monitor DCSS with the Class E privileged CP command Q NSS MAP (the values BEGPAG and ENDPAG are given in units of 4K pages).

See also “CP Command and Utility Reference” (SC24-6081-00) for more information on the DEF STOR and Q NSS MAP commands, as well as “Saved Segments Planning and Administration” (SC24-6116-00) for more information on DCSSes.

4.2.1 1st option:

You can use the CP command DEF STOR CONFIG to define a "memory hole" in your guest virtual storage around the address range of the DCSS.

Example: DEF STOR CONFIG 0.140M 200M.200M

This defines two blocks of storage, the first is 140MB in size an begins at address 0MB, the second is 200MB in size and begins at address 200MB, resulting in a total storage of 340MB. Note that the first block should always start at 0 and be at least 64MB in size.

4.2.2 2nd option:

Your guest virtual storage has to end below the starting address of the DCSS and you have to specify the "mem=" kernel parameter in your parmfile with a value greater than the ending address of the DCSS.

Example:

```
DEF STOR 140M
```

This defines 140MB storage size for your guest, the parameter "mem=160M" is added to the parmfile.

4.3 User Interface

The char device is implemented as a kernel module named "monreader", which can be loaded via the modprobe command, or it can be compiled into the kernel instead. There is one optional module (or kernel) parameter, "mondcss", to specify the name of the monitor DCSS. If the module is compiled into the kernel, the kernel parameter"monreader.mondcss=<DCSS NAME>"can be specified in the parmfile.

The default name for the DCSS is "MONDCSS" if none is specified. In case that there are other users already connected to the *MONITOR service (e.g. Performance Toolkit), the monitor DCSS is already defined and you have to use the same DCSS. The CP command Q MONITOR (Class E privileged) shows the name of the monitor DCSS, if already defined, and the users connected to the *MONITOR service. Refer to the "z/VM Performance" book (SC24-6109-00) on how to create a monitor DCSS if your z/VM doesn't have one already, you need Class E privileges to define and save a DCSS.

4.3.1 Example:

```
modprobe monreader mondcss=MYDCSS
```

This loads the module and sets the DCSS name to “MYDCSS”.

4.3.2 NOTE:

This API provides no interface to control the *MONITOR service, e.g. specify which data should be collected. This can be done by the CP command MONITOR (Class E privileged), see “CP Command and Utility Reference”.

4.3.3 Device nodes with udev:

After loading the module, a char device will be created along with the device node /<udev directory>/monreader.

4.3.4 Device nodes without udev:

If your distribution does not support udev, a device node will not be created automatically and you have to create it manually after loading the module. Therefore you need to know the major and minor numbers of the device. These numbers can be found in /sys/class/misc/monreader/dev.

Typing cat /sys/class/misc/monreader/dev will give an output of the form <major>:<minor>. The device node can be created via the mknod command, enter mknod <name> c <major> <minor>, where <name> is the name of the device node to be created.

4.3.5 Example:

```
# modprobe monreader
# cat /sys/class/misc/monreader/dev
10:63
# mknod /dev/monreader c 10 63
```

This loads the module with the default monitor DCSS (MONDCSS) and creates a device node.

4.3.6 File operations:

The following file operations are supported: open, release, read, poll. There are two alternative methods for reading: either non-blocking read in conjunction with polling, or blocking read without polling. IOCTLs are not supported.

4.3.7 Read:

Reading from the device provides a 12 Byte monitor control element (MCE), followed by a set of one or more contiguous monitor records (similar to the output of the CMS utility MONWRITE without the 4K control blocks). The MCE contains information on the type of the following record set (sample/event data), the monitor domains contained within it and the start and end address of the record set in the monitor DCSS. The start and end address can be used to determine the size of the record set, the end address is the address of the last byte of data. The start address is needed to handle "end-of-frame" records correctly (domain 1, record 13), i.e. it can be used to determine the record start offset relative to a 4K page (frame) boundary.

See "Appendix A: *MONITOR" in the "z/VM Performance" document for a description of the monitor control element layout. The layout of the monitor records can be found here (z/VM 5.1): http://www.vm.ibm.com/pubs/mon510/index.html

The layout of the data stream provided by the monreader device is as follows:

```
...
<0 byte read>
<first MCE>              \
<first set of records>    |
...                       |- data set
<last MCE>                |
<last set of records>    /
<0 byte read>
...
```

There may be more than one combination of MCE and corresponding record set within one data set and the end of each data set is indicated by a successful read with a return value of 0 (0 byte read). Any received data must be considered invalid until a complete set was read successfully, including the closing 0 byte read. Therefore you should always read the complete set into a buffer before processing the data.

The maximum size of a data set can be as large as the size of the monitor DCSS, so design the buffer adequately or use dynamic memory allocation. The size of the monitor DCSS will be printed into syslog after loading the module. You can also use the (Class E privileged) CP command Q NSS MAP to list all available segments and information about them.

As with most char devices, error conditions are indicated by returning a negative value for the number of bytes read. In this case, the errno variable indicates the error condition:

EIO: reply failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EFAULT: copy_to_user failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EAGAIN: occurs on a non-blocking read if there is no data available at the moment. There is no data missing or corrupted, just try again or rather use polling for non-blocking reads.

EOVERFLOW: message limit reached, the data read since the last successful read with 0 size is valid but subsequent records may be missing.

In the last case (EOVERFLOW) there may be missing data, in the first two cases (EIO, EFAULT) there will be missing data. It's up to the application if it will continue reading subsequent data or rather exit.

4.3.8 Open:

Only one user is allowed to open the char device. If it is already in use, the open function will fail (return a negative value) and set errno to EBUSY. The open function may also fail if an IUCV connection to the *MONITOR service cannot be established. In this case errno will be set to EIO and an error message with an IPUSER SEVER code will be printed into syslog. The IPUSER SEVER codes are described in the "z/VM Performance" book, Appendix A.

4.3.9 NOTE:

As soon as the device is opened, incoming messages will be accepted and they will account for the message limit, i.e. opening the device without reading from it will provoke the "message limit reached" error (EOVERFLOW error code) eventually.

CHAPTER

FIVE

IBM S390 QDIO ETHERNET DRIVER

5.1 OSA and HiperSockets Bridge Port Support

5.1.1 Uevents

To generate the events the device must be assigned a role of either a primary or a secondary Bridge Port. For more information, see “z/VM Connectivity, SC24-6174” .

When run on an OSA or HiperSockets Bridge Capable Port hardware, and the state of some configured Bridge Port device on the channel changes, a udev event with ACTION=CHANGE is emitted on behalf of the corresponding ccwgroup device. The event has the following attributes:

BRIDGEPORT=statechange indicates that the Bridge Port device changed its state.

ROLE={primary|secondary|none} the role assigned to the port.

STATE={active|standby|inactive} the newly assumed state of the port.

When run on HiperSockets Bridge Capable Port hardware with host address notifications enabled, a udev event with ACTION=CHANGE is emitted. It is emitted on behalf of the corresponding ccwgroup device when a host or a VLAN is registered or unregistered on the network served by the device. The event has the following attributes:

BRIDGEDHOST={reset|register|deregister|abort} host address notifications are started afresh, a new host or VLAN is registered or deregistered on the Bridge Port HiperSockets channel, or address notifications are aborted.

VLAN=numeric-vlan-id VLAN ID on which the event occurred. Not included if no VLAN is involved in the event.

MAC=xx:xx:xx:xx:xx:xx MAC address of the host that is being registered or deregistered from the HiperSockets channel. Not reported if the event reports the creation or destruction of a VLAN.

NTOK_BUSID=x.y.zzzz device bus ID (CSSID, SSID and device number).

NTOK_IID=xx device IID.

NTOK_CHPID=xx device CHPID.

NTOK_CHID=xxxx device channel ID.

Note that the NTOK_* attributes refer to devices other than the one connected to the system on which the OS is running.

CHAPTER

SIX

S390 DEBUG FEATURE

files:

- arch/s390/kernel/debug.c
- arch/s390/include/asm/debug.h

6.1 Description:

The goal of this feature is to provide a kernel debug logging API where log records can be stored efficiently in memory, where each component (e.g. device drivers) can have one separate debug log. One purpose of this is to inspect the debug logs after a production system crash in order to analyze the reason for the crash.

If the system still runs but only a subcomponent which uses dbf fails, it is possible to look at the debug logs on a live system via the Linux debugfs filesystem.

The debug feature may also very useful for kernel and driver development.

6.2 Design:

Kernel components (e.g. device drivers) can register themselves at the debug feature with the function call `debug_register()`. This function initializes a debug log for the caller. For each debug log exists a number of debug areas where exactly one is active at one time. Each debug area consists of contiguous pages in memory. In the debug areas there are stored debug entries (log records) which are written by event- and exception-calls.

An event-call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around is done (ring buffer) and the next debug entry will be written at the beginning of the active debug area.

An exception-call writes the specified debug entry to the log and switches to the next debug area. This is done in order to be sure that the records which describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas themselves are also ordered in form of a ring buffer. When an exception is thrown in the last debug area, the following debug entries are then written again in the very first area.

There are four versions for the event- and exception-calls: One for logging raw data, one for text, one for numbers (unsigned int and long), and one for sprintf-like formatted strings.

Each debug entry contains the following data:

- Timestamp
- Cpu-Number of calling task
- Level of debug entry (0···6)
- Return Address to caller
- Flag, if entry is an exception or not

The debug logs can be inspected in a live system through entries in the debugfs-filesystem. Under the toplevel directory "`s390dbf`" there is a directory for each registered component, which is named like the corresponding component. The debugfs normally should be mounted to `/sys/kernel/debug` therefore the debug feature can be accessed under `/sys/kernel/debug/s390dbf`.

The content of the directories are files which represent different views to the debug log. Each component can decide which views should be used through registering them with the function `debug_register_view()`. Predefined views for hex/ascii, sprintf and raw binary data are provided. It is also possible to define other views. The content of a view can be inspected simply by reading the corresponding debugfs file.

All debug logs have an actual debug level (range from 0 to 6). The default level is 3. Event and Exception functions have a `level` parameter. Only debug entries with a level that is lower or equal than the actual level are written to the log. This means, when writing events, high priority log entries should have a low level value whereas low priority entries should have a high one. The actual debug level can be changed with the help of the debugfs-filesystem through writing a number string "x" to the `level` debugfs file which is provided for every debug log. Debugging can be switched off completely by using "-" on the `level` debugfs file.

Example:

```
> echo "-" > /sys/kernel/debug/s390dbf/dasd/level
```

It is also possible to deactivate the debug feature globally for every debug log. You can change the behavior using 2 sysctl parameters in `/proc/sys/s390dbf`:

There are currently 2 possible triggers, which stop the debug feature globally. The first possibility is to use the `debug_active` sysctl. If set to 1 the debug feature is running. If `debug_active` is set to 0 the debug feature is turned off.

The second trigger which stops the debug feature is a kernel oops. That prevents the debug feature from overwriting debug information that happened before the oops. After an oops you can reactivate the debug feature by piping 1 to `/proc/sys/s390dbf/debug_active`. Nevertheless, it's not suggested to use an oopsed kernel in a production environment.

If you want to disallow the deactivation of the debug feature, you can use the `debug_stoppable` sysctl. If you set `debug_stoppable` to 0 the debug feature cannot be stopped. If the debug feature is already stopped, it will stay deactivated.

6.3 Kernel Interfaces:

debug_info_t * **debug_register_mode**(const char * name,
int pages_per_area, int nr_areas,
int buf_size, umode_t mode, uid_t uid,
gid_t gid)
creates and initializes debug area.

Parameters

const char * name Name of debug log (e.g. used for debugfs entry)

int pages_per_area Number of pages, which will be allocated per area

int nr_areas Number of debug areas

int buf_size Size of data area in each debug entry

umode_t mode File mode for debugfs files. E.g. S_IRWXUGO

uid_t uid User ID for debugfs files. Currently only 0 is supported.

gid_t gid Group ID for debugfs files. Currently only 0 is supported.

Return

- Handle for generated debug area
- `NULL` if register failed

Description

Allocates memory for a debug log. Must not be called within an interrupt handler.

debug_info_t * **debug_register**(const char * name, int pages_per_area,
int nr_areas, int buf_size)
creates and initializes debug area with default file mode.

Parameters

const char * name Name of debug log (e.g. used for debugfs entry)

int pages_per_area Number of pages, which will be allocated per area

int nr_areas Number of debug areas

int buf_size Size of data area in each debug entry

Return

- Handle for generated debug area
- `NULL` if register failed

Description

Allocates memory for a debug log. The debugfs file mode access permissions are read and write for user. Must not be called within an interrupt handler.

void **debug_unregister**(debug_info_t * id)
give back debug area.

Parameters

debug_info_t * id handle for debug log

Return

none

void **debug_set_level**(debug_info_t * id, int new_level)
Sets new actual debug level if new_level is valid.

Parameters

debug_info_t * id handle for debug log

int new_level new debug level

Return

none

void **debug_stop_all**(void)
stops the debug feature if stopping is allowed.

Parameters

void no arguments

Return

- none

Description

Currently used in case of a kernel oops.

void **debug_set_critical**(void)
event/exception functions try lock instead of spin.

Parameters

void no arguments

Return

- none

Description

Currently used in case of stopping all CPUs but the current one. Once in this state, functions to write a debug entry for an event or exception no longer spin on the debug area lock, but only try to get it and fail if they do not get the lock.

int **debug_register_view**(debug_info_t * id, struct debug_view * view)
registers new debug view and creates debugfs dir entry

Parameters

debug_info_t * id handle for debug log

struct debug_view * view pointer to debug view struct

Return

- 0 : ok
- < 0: Error

int **debug_unregister_view**(debug_info_t * id, struct debug_view * view)
unregisters debug view and removes debugfs dir entry

Parameters

debug_info_t * id handle for debug log

struct debug_view * view pointer to debug view struct

Return

- 0 : ok
- < 0: Error

bool **debug_level_enabled**(debug_info_t * id, int level)
Returns true if debug events for the specified level would be logged. Otherwise returns false.

Parameters

debug_info_t * id handle for debug log

int level debug level

Return

- `true` if level is less or equal to the current debug level.

debug_entry_t * **debug_event**(debug_info_t * id, int level, void * data, int length)
writes binary debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t * id handle for debug log

int level debug level

void * data pointer to data for debug entry

int length length of data in bytes

Return

- Address of written debug entry
- `NULL` if error

debug_entry_t * **debug_int_event**(debug_info_t * id, int level, unsigned int tag)
writes unsigned integer debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t * id handle for debug log

int level debug level

unsigned int tag integer value for debug entry

Return

- Address of written debug entry

- NULL if error

debug_entry_t * **debug_long_event**(debug_info_t * id, int level, unsigned long tag)

writes unsigned long debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t * id handle for debug log

int level debug level

unsigned long tag long integer value for debug entry

Return

- Address of written debug entry
- NULL if error

debug_entry_t * **debug_text_event**(debug_info_t * id, int level, const char * txt)

writes string debug entry in ascii format to active debug area (if level <= actual debug level)

Parameters

debug_info_t * id handle for debug log

int level debug level

const char * txt string for debug entry

Return

- Address of written debug entry
- NULL if error

debug_sprintf_event(_id, _level, _fmt, ⋯)

writes debug entry with format string and varargs (longs) to active debug area (if level $<=$ actual debug level).

Parameters

_id handle for debug log

_level debug level

_fmt format string for debug entry

... varargs used as in sprintf()

Return

- Address of written debug entry
- NULL if error

Description

floats and long long datatypes cannot be used as varargs.

debug_entry_t * **debug_exception**(debug_info_t * id, int level, void * data, int length)

writes binary debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

debug_info_t * id handle for debug log

int level debug level

void * data pointer to data for debug entry

int length length of data in bytes

Return

- Address of written debug entry
- NULL if error

debug_entry_t * **debug_int_exception**(debug_info_t * id, int level, unsigned int tag)

writes unsigned int debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

debug_info_t * id handle for debug log

int level debug level

unsigned int tag integer value for debug entry

Return

- Address of written debug entry
- NULL if error

debug_entry_t * **debug_long_exception**(debug_info_t * id, int level, unsigned long tag)

writes long debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

debug_info_t * id handle for debug log

int level debug level

unsigned long tag long integer value for debug entry

Return

- Address of written debug entry
- NULL if error

debug_entry_t * **debug_text_exception**(debug_info_t * id, int level, const char * txt)

writes string debug entry in ascii format to active debug area (if level <= actual debug level) and switches to next debug area area

Parameters

debug_info_t * id handle for debug log

int level debug level

const char * txt string for debug entry

Return

- Address of written debug entry
- NULL if error

debug_sprintf_exception(_id, _level, _fmt, ...)
writes debug entry with format string and varargs (longs) to active debug area (if level <= actual debug level) and switches to next debug area.

Parameters

_id handle for debug log

_level debug level

_fmt format string for debug entry

... varargs used as in sprintf()

Return

- Address of written debug entry
- NULL if error

Description

floats and long long datatypes cannot be used as varargs.

6.4 Predefined views:

```
extern struct debug_view debug_hex_ascii_view;

extern struct debug_view debug_raw_view;

extern struct debug_view debug_sprintf_view;
```

6.5 Examples

```
/*
 * hex_ascii- + raw-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t *debug_info;

static int init(void)
{
```

(continues on next page)

(continued from previous page)

```
    /* register 4 debug areas with one page each and 4 byte data field */

    debug_info = debug_register("test", 1, 4, 4 );
    debug_register_view(debug_info, &debug_hex_ascii_view);
    debug_register_view(debug_info, &debug_raw_view);

    debug_text_event(debug_info, 4 , "one ");
    debug_int_exception(debug_info, 4, 4711);
    debug_event(debug_info, 3, &debug_info, 4);

    return 0;
}

static void cleanup(void)
{
    debug_unregister(debug_info);
}

module_init(init);
module_exit(cleanup);
```

```
/*
 * sprintf-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t *debug_info;

static int init(void)
{
    /* register 4 debug areas with one page each and data field for */
    /* format string pointer + 2 varargs (= 3 * sizeof(long))       */

    debug_info = debug_register("test", 1, 4, sizeof(long) * 3);
    debug_register_view(debug_info, &debug_sprintf_view);

    debug_sprintf_event(debug_info, 2 , "first event in %s:%i\n",__FILE__,_
↪_LINE__);
    debug_sprintf_exception(debug_info, 1, "pointer to debug info: %p\n",&
↪debug_info);

    return 0;
}

static void cleanup(void)
{
    debug_unregister(debug_info);
}

module_init(init);
module_exit(cleanup);
```

6.6 Debugfs Interface

Views to the debug logs can be investigated through reading the corresponding debugfs-files:

Example:

```
> ls /sys/kernel/debug/s390dbf/dasd
flush  hex_ascii  level pages raw
> cat /sys/kernel/debug/s390dbf/dasd/hex_ascii | sort -k2,2 -s
00 00974733272:680099 2 - 02 0006ad7e  07 ea 4a 90 | ....
00 00974733272:682210 2 - 02 0006ade6  46 52 45 45 | FREE
00 00974733272:682213 2 - 02 0006adf6  07 ea 4a 90 | ....
00 00974733272:682281 1 * 02 0006ab08  41 4c 4c 43 | EXCP
01 00974733272:682284 2 - 02 0006ab16  45 43 4b 44 | ECKD
01 00974733272:682287 2 - 02 0006ab28  00 00 00 04 | ....
01 00974733272:682289 2 - 02 0006ab3e  00 00 00 20 | ...
01 00974733272:682297 2 - 02 0006ad7e  07 ea 4a 90 | ....
01 00974733272:684384 2 - 00 0006ade6  46 52 45 45 | FREE
01 00974733272:684388 2 - 00 0006adf6  07 ea 4a 90 | ....
```

See section about predefined views for explanation of the above output!

6.7 Changing the debug level

Example:

```
> cat /sys/kernel/debug/s390dbf/dasd/level
3
> echo "5" > /sys/kernel/debug/s390dbf/dasd/level
> cat /sys/kernel/debug/s390dbf/dasd/level
5
```

6.8 Flushing debug areas

Debug areas can be flushed with piping the number of the desired area (0⋯n) to the debugfs file “flush”. When using “-” all debug areas are flushed.

Examples:

1. Flush debug area 0:

   ```
   > echo "0" > /sys/kernel/debug/s390dbf/dasd/flush
   ```

2. Flush all debug areas:

   ```
   > echo "-" > /sys/kernel/debug/s390dbf/dasd/flush
   ```

6.9 Changing the size of debug areas

It is possible the change the size of debug areas through piping the number of pages to the debugfs file “pages”. The resize request will also flush the debug areas.

Example:

Define 4 pages for the debug areas of debug feature “dasd”:

```
> echo "4" > /sys/kernel/debug/s390dbf/dasd/pages
```

6.10 Stopping the debug feature

Example:

1. Check if stopping is allowed:

   ```
   > cat /proc/sys/s390dbf/debug_stoppable
   ```

2. Stop debug feature:

   ```
   > echo 0 > /proc/sys/s390dbf/debug_active
   ```

6.11 crash Interface

The `crash` tool since v5.1.0 has a built-in command `s390dbf` to display all the debug logs or export them to the file system. With this tool it is possible to investigate the debug logs on a live system and with a memory dump after a system crash.

6.12 Investigating raw memory

One last possibility to investigate the debug logs at a live system and after a system crash is to look at the raw memory under VM or at the Service Element. It is possible to find the anchor of the debug-logs through the `debug_area_first` symbol in the System map. Then one has to follow the correct pointers of the data-structures defined in debug.h and find the debug-areas in memory. Normally modules which use the debug feature will also have a global variable with the pointer to the debug-logs. Following this pointer it will also be possible to find the debug logs in memory.

For this method it is recommended to use ‘16 * x + 4’ byte (x = 0..n) for the length of the data field in `debug_register()` in order to see the debug entries well formatted.

6.13 Predefined Views

There are three predefined views: hex_ascii, raw and sprintf. The hex_ascii view shows the data field in hex and ascii representation (e.g. `45 43 4b 44 | ECKD`). The raw view returns a bytestream as the debug areas are stored in memory.

The sprintf view formats the debug entries in the same way as the sprintf function would do. The sprintf event/exception functions write to the debug entry a pointer to the format string (size = sizeof(long)) and for each vararg a long value. So e.g. for a debug entry with a format string plus two varargs one would need to allocate a (3 * sizeof(long)) byte data area in the `debug_register()` function.

IMPORTANT: Using "%s" in sprintf event functions is dangerous. You can only use "%s" in the sprintf event functions, if the memory for the passed string is available as long as the debug feature exists. The reason behind this is that due to performance considerations only a pointer to the string is stored in the debug feature. If you log a string that is freed afterwards, you will get an OOPS when inspecting the debug feature, because then the debug feature will access the already freed memory.

NOTE: If using the sprintf view do NOT use other event/exception functions than the sprintf-event and -exception functions.

The format of the hex_ascii and sprintf view is as follows:

- Number of area
- Timestamp (formatted as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970)
- level of debug entry
- Exception flag (* = Exception)
- Cpu-Number of calling task
- Return Address to caller
- data field

The format of the raw view is:

- Header as described in debug.h
- datafield

A typical line of the hex_ascii view will look like the following (first line is only for explanation and will not be displayed when 'cating' the view):

```
area  time           level exception cpu caller    data (hex + ascii)
------------------------------------------------------------------------
00    00964419409:440690 1 -         00  88023fe
```

6.14 Defining views

Views are specified with the ‘debug_view’ structure. There are defined callback functions which are used for reading and writing the debugfs files:

```
struct debug_view {
      char name[DEBUG_MAX_PROCF_LEN];
      debug_prolog_proc_t* prolog_proc;
      debug_header_proc_t* header_proc;
      debug_format_proc_t* format_proc;
      debug_input_proc_t*  input_proc;
      void*                private_data;
};
```

where:

```
typedef int (debug_header_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   int area,
                                   debug_entry_t* entry,
                                   char* out_buf);

typedef int (debug_format_proc_t) (debug_info_t* id,
                                   struct debug_view* view, char* out_buf,
                                   const char* in_buf);
typedef int (debug_prolog_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   char* out_buf);
typedef int (debug_input_proc_t) (debug_info_t* id,
                                  struct debug_view* view,
                                  struct file* file, const char* user_buf,
                                  size_t in_buf_size, loff_t* offset);
```

The “private_data” member can be used as pointer to view specific data. It is not used by the debug feature itself.

The output when reading a debugfs file is structured like this:

```
"prolog_proc output"

"header_proc output 1"  "format_proc output 1"
"header_proc output 2"  "format_proc output 2"
"header_proc output 3"  "format_proc output 3"
...
```

When a view is read from the debugfs, the Debug Feature calls the ‘prolog_proc’ once for writing the prolog. Then ‘header_proc’ and ‘format_proc’ are called for each existing debug entry.

The input_proc can be used to implement functionality when it is written to the view (e.g. like with `echo "0" > /sys/kernel/debug/s390dbf/dasd/level`).

For header_proc there can be used the default function `debug_dflt_header_fn()` which is defined in debug.h. and which produces the same header output as the predefined views. E.g:

```
00 00964419409:440761 2 - 00 88023ec
```

In order to see how to use the callback functions check the implementation of the default views!

Example:

```
#include <asm/debug.h>

#define UNKNOWNSTR "data: %08x"

const char* messages[] =
{"This error...........\n",
 "That error...........\n",
 "Problem..............\n",
 "Something went wrong.\n",
 "Everything ok........\n",
 NULL
};

static int debug_test_format_fn(
   debug_info_t *id, struct debug_view *view,
   char *out_buf, const char *in_buf
)
{
  int i, rc = 0;

  if (id->buf_size >= 4) {
     int msg_nr = *((int*)in_buf);
     if (msg_nr < sizeof(messages) / sizeof(char*) - 1)
        rc += sprintf(out_buf, "%s", messages[msg_nr]);
     else
        rc += sprintf(out_buf, UNKNOWNSTR, msg_nr);
  }
  return rc;
}

struct debug_view debug_test_view = {
  "myview",                   /* name of view */
  NULL,                       /* no prolog */
  &debug_dflt_header_fn,      /* default header for each entry */
  &debug_test_format_fn,      /* our own format function */
  NULL,                       /* no input function */
  NULL                        /* no private data */
};
```

6.14.1 test:

```
debug_info_t *debug_info;
int i;
...
debug_info = debug_register("test", 0, 4, 4);
debug_register_view(debug_info, &debug_test_view);
for (i = 0; i < 10; i ++)
  debug_int_event(debug_info, 1, i);
```

```
> cat /sys/kernel/debug/s390dbf/test/myview
00 00964419734:611402 1 - 00 88042ca   This error.....
00 00964419734:611405 1 - 00 88042ca   That error.....
00 00964419734:611408 1 - 00 88042ca   Problem.....
00 00964419734:611411 1 - 00 88042ca   Something went wrong.
00 00964419734:611414 1 - 00 88042ca   Everything ok........
00 00964419734:611417 1 - 00 88042ca   data: 00000005
00 00964419734:611419 1 - 00 88042ca   data: 00000006
00 00964419734:611422 1 - 00 88042ca   data: 00000007
00 00964419734:611425 1 - 00 88042ca   data: 00000008
00 00964419734:611428 1 - 00 88042ca   data: 00000009
```

CHAPTER

SEVEN

ADJUNCT PROCESSOR (AP) FACILITY

7.1 Introduction

The Adjunct Processor (AP) facility is an IBM Z cryptographic facility comprised of three AP instructions and from 1 up to 256 PCIe cryptographic adapter cards. The AP devices provide cryptographic functions to all CPUs assigned to a linux system running in an IBM Z system LPAR.

The AP adapter cards are exposed via the AP bus. The motivation for vfio-ap is to make AP cards available to KVM guests using the VFIO mediated device framework. This implementation relies considerably on the s390 virtualization facilities which do most of the hard work of providing direct access to AP devices.

7.2 AP Architectural Overview

To facilitate the comprehension of the design, let's start with some definitions:

- AP adapter

  An AP adapter is an IBM Z adapter card that can perform cryptographic functions. There can be from 0 to 256 adapters assigned to an LPAR. Adapters assigned to the LPAR in which a linux host is running will be available to the linux host. Each adapter is identified by a number from 0 to 255; however, the maximum adapter number is determined by machine model and/or adapter type. When installed, an AP adapter is accessed by AP instructions executed by any CPU.

  The AP adapter cards are assigned to a given LPAR via the system's Activation Profile which can be edited via the HMC. When the linux host system is IPL'd in the LPAR, the AP bus detects the AP adapter cards assigned to the LPAR and creates a sysfs device for each assigned adapter. For example, if AP adapters 4 and 10 (0x0a) are assigned to the LPAR, the AP bus will create the following sysfs device entries:

  ```
  /sys/devices/ap/card04
  /sys/devices/ap/card0a
  ```

  Symbolic links to these devices will also be created in the AP bus devices sub-directory:

```
/sys/bus/ap/devices/[card04]
/sys/bus/ap/devices/[card04]
```

- AP domain

  An adapter is partitioned into domains. An adapter can hold up to 256 domains depending upon the adapter type and hardware configuration. A domain is identified by a number from 0 to 255; however, the maximum domain number is determined by machine model and/or adapter type.. A domain can be thought of as a set of hardware registers and memory used for processing AP commands. A domain can be configured with a secure private key used for clear key encryption. A domain is classified in one of two ways depending upon how it may be accessed:

  - Usage domains are domains that are targeted by an AP instruction to process an AP command.
  - Control domains are domains that are changed by an AP command sent to a usage domain; for example, to set the secure private key for the control domain.

  The AP usage and control domains are assigned to a given LPAR via the system's Activation Profile which can be edited via the HMC. When a linux host system is IPL'd in the LPAR, the AP bus module detects the AP usage and control domains assigned to the LPAR. The domain number of each usage domain and adapter number of each AP adapter are combined to create AP queue devices (see AP Queue section below). The domain number of each control domain will be represented in a bitmask and stored in a sysfs file /sys/bus/ap/ap_control_domain_mask. The bits in the mask, from most to least significant bit, correspond to domains 0-255.

- AP Queue

  An AP queue is the means by which an AP command is sent to a usage domain inside a specific adapter. An AP queue is identified by a tuple comprised of an AP adapter ID (APID) and an AP queue index (APQI). The APQI corresponds to a given usage domain number within the adapter. This tuple forms an AP Queue Number (APQN) uniquely identifying an AP queue. AP instructions include a field containing the APQN to identify the AP queue to which the AP command is to be sent for processing.

  The AP bus will create a sysfs device for each APQN that can be derived from the cross product of the AP adapter and usage domain numbers detected when the AP bus module is loaded. For example, if adapters 4 and 10 (0x0a) and usage domains 6 and 71 (0x47) are assigned to the LPAR, the AP bus will create the following sysfs entries:

  ```
  /sys/devices/ap/card04/04.0006
  /sys/devices/ap/card04/04.0047
  /sys/devices/ap/card0a/0a.0006
  /sys/devices/ap/card0a/0a.0047
  ```

  The following symbolic links to these devices will be created in the AP bus devices subdirectory:

```
/sys/bus/ap/devices/[04.0006]
/sys/bus/ap/devices/[04.0047]
/sys/bus/ap/devices/[0a.0006]
/sys/bus/ap/devices/[0a.0047]
```

- AP Instructions:

  There are three AP instructions:

  - NQAP: to enqueue an AP command-request message to a queue
  - DQAP: to dequeue an AP command-reply message from a queue
  - PQAP: to administer the queues

  AP instructions identify the domain that is targeted to process the AP command; this must be one of the usage domains. An AP command may modify a domain that is not one of the usage domains, but the modified domain must be one of the control domains.

7.3 AP and SIE

Let's now take a look at how AP instructions executed on a guest are interpreted by the hardware.

A satellite control block called the Crypto Control Block (CRYCB) is attached to our main hardware virtualization control block. The CRYCB contains three fields to identify the adapters, usage domains and control domains assigned to the KVM guest:

- The AP Mask (APM) field is a bit mask that identifies the AP adapters assigned to the KVM guest. Each bit in the mask, from left to right (i.e. from most significant to least significant bit in big endian order), corresponds to an APID from 0-255. If a bit is set, the corresponding adapter is valid for use by the KVM guest.
- The AP Queue Mask (AQM) field is a bit mask identifying the AP usage domains assigned to the KVM guest. Each bit in the mask, from left to right (i.e. from most significant to least significant bit in big endian order), corresponds to an AP queue index (APQI) from 0-255. If a bit is set, the corresponding queue is valid for use by the KVM guest.
- The AP Domain Mask field is a bit mask that identifies the AP control domains assigned to the KVM guest. The ADM bit mask controls which domains can be changed by an AP command-request message sent to a usage domain from the guest. Each bit in the mask, from left to right (i.e. from most significant to least significant bit in big endian order), corresponds to a domain from 0-255. If a bit is set, the corresponding domain can be modified by an AP command-request message sent to a usage domain.

If you recall from the description of an AP Queue, AP instructions include an APQN to identify the AP queue to which an AP command-request message is to be sent (NQAP and PQAP instructions), or from which a command-reply message is to be received (DQAP instruction). The validity of an APQN is defined by the matrix calculated from the APM and AQM; it is the cross product of all assigned adapter

numbers (APM) with all assigned queue indexes (AQM). For example, if adapters 1 and 2 and usage domains 5 and 6 are assigned to a guest, the APQNs (1,5), (1,6), (2,5) and (2,6) will be valid for the guest.

The APQNs can provide secure key functionality - i.e., a private key is stored on the adapter card for each of its domains - so each APQN must be assigned to at most one guest or to the linux host:

```
Example 1: Valid configuration:
-------------------------------
Guest1: adapters 1,2  domains 5,6
Guest2: adapter  1,2  domain 7

This is valid because both guests have a unique set of APQNs:
   Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
   Guest2 has APQNs (1,7), (2,7)

Example 2: Valid configuration:
-------------------------------
Guest1: adapters 1,2 domains 5,6
Guest2: adapters 3,4 domains 5,6

This is also valid because both guests have a unique set of APQNs:
   Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
   Guest2 has APQNs (3,5), (3,6), (4,5), (4,6)

Example 3: Invalid configuration:
---------------------------------
Guest1: adapters 1,2  domains 5,6
Guest2: adapter  1    domains 6,7

This is an invalid configuration because both guests have access to
APQN (1,6).
```

7.4 The Design

The design introduces three new objects:

1. AP matrix device
2. VFIO AP device driver (vfio_ap.ko)
3. VFIO AP mediated matrix pass-through device

7.4.1 The VFIO AP device driver

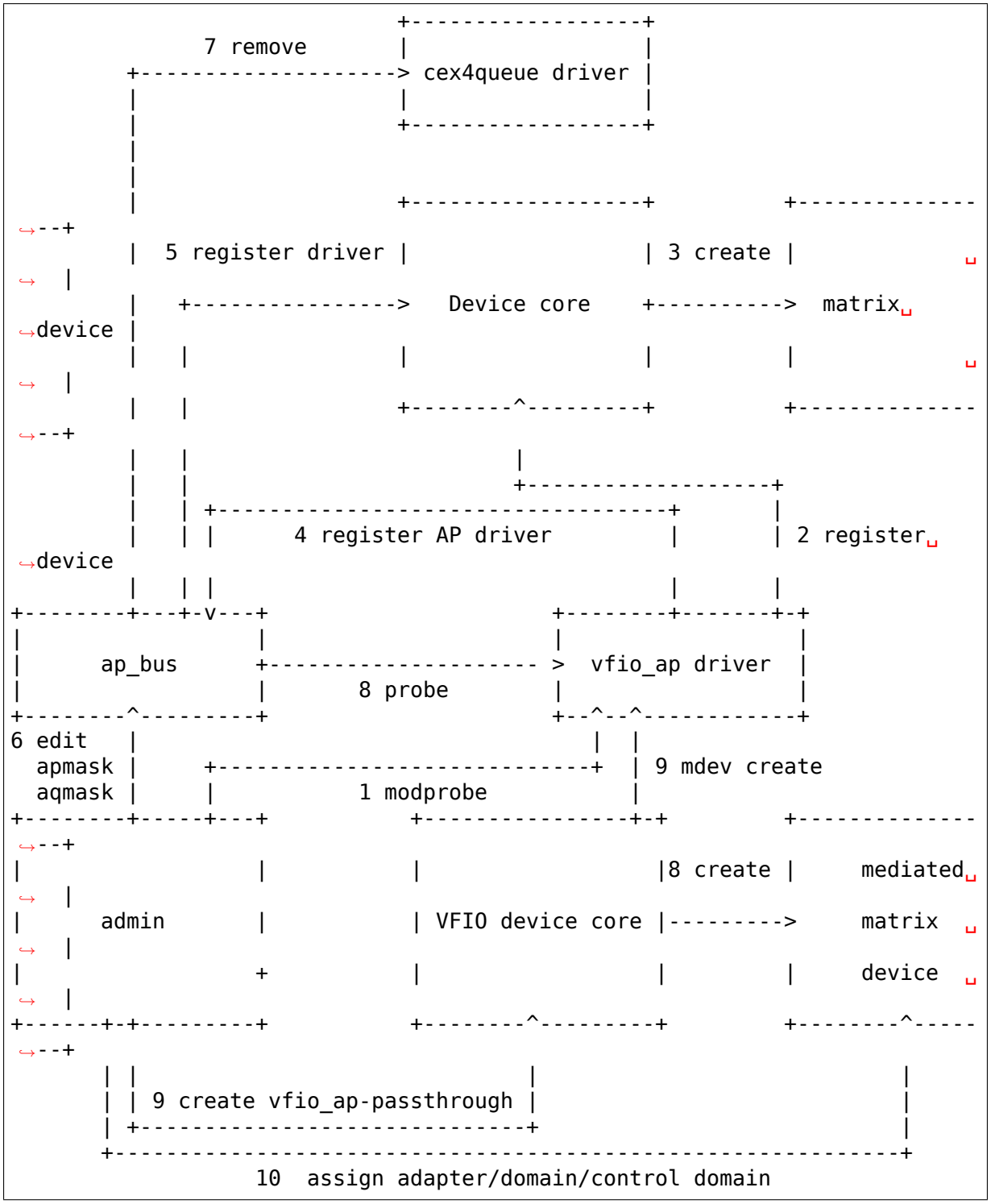
The VFIO AP (vfio_ap) device driver serves the following purposes:

1. Provides the interfaces to secure APQNs for exclusive use of KVM guests.
2. Sets up the VFIO mediated device interfaces to manage a mediated matrix device and creates the sysfs interfaces for assigning adapters, usage domains, and control domains comprising the matrix for a KVM guest.

3. Configures the APM, AQM and ADM in the CRYCB referenced by a KVM guest's SIE state description to grant the guest access to a matrix of AP devices

7.4.2 Reserve APQNs for exclusive use of KVM guests

The following block diagram illustrates the mechanism by which APQNs are reserved:



The process for reserving an AP queue for use by a KVM guest is:

1. The administrator loads the vfio_ap device driver
2. The vfio-ap driver during its initialization will register a single 'matrix' device with the device core. This will serve as the parent device for all mediated matrix devices used to configure an AP matrix for a guest.
3. The /sys/devices/vfio_ap/matrix device is created by the device core
4. The vfio_ap device driver will register with the AP bus for AP queue devices of type 10 and higher (CEX4 and newer). The driver will provide the vfio_ap driver's probe and remove callback interfaces. Devices older than CEX4 queues are not supported to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future, and for which there are few older systems around on which to test.
5. The AP bus registers the vfio_ap device driver with the device core
6. The administrator edits the AP adapter and queue masks to reserve AP queues for use by the vfio_ap device driver.
7. The AP bus removes the AP queues reserved for the vfio_ap driver from the default zcrypt cex4queue driver.
8. The AP bus probes the vfio_ap device driver to bind the queues reserved for it.
9. The administrator creates a passthrough type mediated matrix device to be used by a guest
10. The administrator assigns the adapters, usage domains and control domains to be exclusively used by a guest.

7.4.3 Set up the VFIO mediated device interfaces

The VFIO AP device driver utilizes the common interface of the VFIO mediated device core driver to:

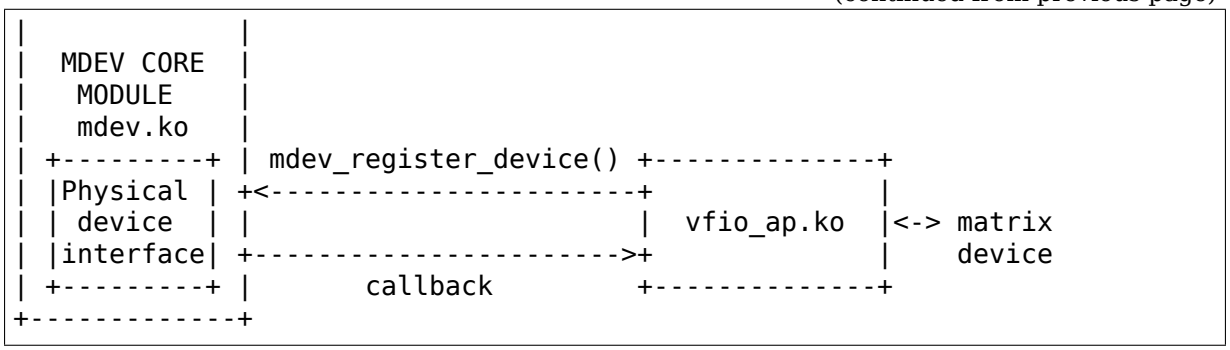
- Register an AP mediated bus driver to add a mediated matrix device to and remove it from a VFIO group.
- Create and destroy a mediated matrix device
- Add a mediated matrix device to and remove it from the AP mediated bus driver
- Add a mediated matrix device to and remove it from an IOMMU group

The following high-level block diagram shows the main components and interfaces of the VFIO AP mediated matrix device driver:

```
+-------------+
|             |
| +---------+ | mdev_register_driver() +--------------+
| |  Mdev   | +<-----------------------+              |
| |  bus    | |                        | vfio_mdev.ko |
| | driver  | +----------------------->+              |<-> VFIO user
| +---------+ |    probe()/remove()    +--------------+    APIs
```

(continues on next page)

(continued from previous page)



During initialization of the vfio_ap module, the matrix device is registered with an 'mdev_parent_ops' structure that provides the sysfs attribute structures, mdev functions and callback interfaces for managing the mediated matrix device.

- sysfs attribute structures:

  supported_type_groups The VFIO mediated device framework supports creation of user-defined mediated device types. These mediated device types are specified via the 'supported_type_groups' structure when a device is registered with the mediated device framework. The registration process creates the sysfs structures for each mediated device type specified in the 'mdev_supported_types' sub-directory of the device being registered. Along with the device type, the sysfs attributes of the mediated device type are provided.

  The VFIO AP device driver will register one mediated device type for passthrough devices:

  > /sys/devices/vfio_ap/matrix/mdev_supported_types/vfio_ap-passthrough

  Only the read-only attributes required by the VFIO mdev framework will be provided:

  ```
  ... name
  ... device_api
  ... available_instances
  ... device_api
  ```

  Where:

  - **name:** specifies the name of the mediated device type
  - **device_api:** the mediated device type's API
  - **available_instances:** the number of mediated matrix passthrough devices that can be created
  - **device_api:** specifies the VFIO API

  mdev_attr_groups This attribute group identifies the user-defined sysfs attributes of the mediated device. When a device is registered with the VFIO mediated device framework, the sysfs attribute files identified in the 'mdev_attr_groups' structure will be created in the mediated matrix device's directory. The sysfs attributes for a mediated matrix device are:

assign_adapter / unassign_adapter: Write-only attributes for assigning/unassigning an AP adapter to/from the mediated matrix device. To assign/unassign an adapter, the APID of the adapter is echoed to the respective attribute file.

assign_domain / unassign_domain: Write-only attributes for assigning/unassigning an AP usage domain to/from the mediated matrix device. To assign/unassign a domain, the domain number of the the usage domain is echoed to the respective attribute file.

matrix: A read-only file for displaying the APQNs derived from the cross product of the adapter and domain numbers assigned to the mediated matrix device.

assign_control_domain / unassign_control_domain: Write-only attributes for assigning/unassigning an AP control domain to/from the mediated matrix device. To assign/unassign a control domain, the ID of the domain to be assigned/unassigned is echoed to the respective attribute file.

control_domains: A read-only file for displaying the control domain numbers assigned to the mediated matrix device.

- functions:

  create: allocates the ap_matrix_mdev structure used by the vfio_ap driver to:

  - Store the reference to the KVM structure for the guest using the mdev
  - Store the AP matrix configuration for the adapters, domains, and control domains assigned via the corresponding sysfs attributes files

  remove: deallocates the mediated matrix device's ap_matrix_mdev structure. This will be allowed only if a running guest is not using the mdev.

- callback interfaces

  open: The vfio_ap driver uses this callback to register a VFIO_GROUP_NOTIFY_SET_KVM notifier callback function for the mdev matrix device. The open is invoked when QEMU connects the VFIO iommu group for the mdev matrix device to the MDEV bus. Access to the KVM structure used to configure the KVM guest is provided via this callback. The KVM structure, is used to configure the guest's access to the AP matrix defined via the mediated matrix device's sysfs attribute files.

  release: unregisters the VFIO_GROUP_NOTIFY_SET_KVM notifier callback function for the mdev matrix device and deconfigures the guest's AP matrix.

7.4.4 Configure the APM, AQM and ADM in the CRYCB

Configuring the AP matrix for a KVM guest will be performed when the VFIO_GROUP_NOTIFY_SET_KVM notifier callback is invoked. The notifier function is called when QEMU connects to KVM. The guest's AP matrix is configured via it's CRYCB by:

- Setting the bits in the APM corresponding to the APIDs assigned to the mediated matrix device via its 'assign_adapter' interface.
- Setting the bits in the AQM corresponding to the domains assigned to the mediated matrix device via its 'assign_domain' interface.
- Setting the bits in the ADM corresponding to the domain dIDs assigned to the mediated matrix device via its 'assign_control_domains' interface.

7.4.5 The CPU model features for AP

The AP stack relies on the presence of the AP instructions as well as two facilities: The AP Facilities Test (APFT) facility; and the AP Query Configuration Information (QCI) facility. These features/facilities are made available to a KVM guest via the following CPU model features:

1. ap: Indicates whether the AP instructions are installed on the guest. This feature will be enabled by KVM only if the AP instructions are installed on the host.
2. apft: Indicates the APFT facility is available on the guest. This facility can be made available to the guest only if it is available on the host (i.e., facility bit 15 is set).
3. apqci: Indicates the AP QCI facility is available on the guest. This facility can be made available to the guest only if it is available on the host (i.e., facility bit 12 is set).

Note: If the user chooses to specify a CPU model different than the 'host' model to QEMU, the CPU model features and facilities need to be turned on explicitly; for example:

```
/usr/bin/qemu-system-s390x ... -cpu z13,ap=on,apqci=on,apft=on
```

A guest can be precluded from using AP features/facilities by turning them off explicitly; for example:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=off,apqci=off,apft=off
```

Note: If the APFT facility is turned off (apft=off) for the guest, the guest will not see any AP devices. The zcrypt device drivers that register for type 10 and newer AP devices - i.e., the cex4card and cex4queue device drivers - need the APFT facility to ascertain the facilities installed on a given AP device. If the APFT facility is not installed on the guest, then the probe of device drivers will fail since only type 10 and newer devices can be configured for guest use.

7.5 Example

Let's now provide an example to illustrate how KVM guests may be given access to AP facilities. For this example, we will show how to configure three guests such that executing the lszcrypt command on the guests would look like this:

7.5.1 Guest1

CARD.DOMAIN	TYPE	MODE
05	CEX5C	CCA-Coproc
05.0004	CEX5C	CCA-Coproc
05.00ab	CEX5C	CCA-Coproc
06	CEX5A	Accelerator
06.0004	CEX5A	Accelerator
06.00ab	CEX5C	CCA-Coproc

7.5.2 Guest2

CARD.DOMAIN	TYPE	MODE
05	CEX5A	Accelerator
05.0047	CEX5A	Accelerator
05.00ff	CEX5A	Accelerator

7.5.3 Guest3

CARD.DOMAIN	TYPE	MODE
06	CEX5A	Accelerator
06.0047	CEX5A	Accelerator
06.00ff	CEX5A	Accelerator

These are the steps:

1. Install the vfio_ap module on the linux host. The dependency chain for the vfio_ap module is: * iommu * s390 * zcrypt * vfio * vfio_mdev * vfio_mdev_device * KVM

   To build the vfio_ap module, the kernel build must be configured with the following Kconfig elements selected: * IOMMU_SUPPORT * S390 * ZCRYPT * S390_AP_IOMMU * VFIO * VFIO_MDEV * VFIO_MDEV_DEVICE * KVM

   If using make menuconfig select the following to build the vfio_ap module:

```
-> Device Drivers
   -> IOMMU Hardware Support
      select S390 AP IOMMU Support
   -> VFIO Non-Privileged userspace driver framework
      -> Mediated device driver frramework
```

(continues on next page)

(continued from previous page)

```
        -> VFIO driver for Mediated devices
-> I/O subsystem
   -> VFIO support for AP devices
```

2. Secure the AP queues to be used by the three guests so that the host can not access them. To secure them, there are two sysfs files that specify bitmasks marking a subset of the APQN range as 'usable by the default AP queue device drivers' or 'not usable by the default device drivers' and thus available for use by the vfio_ap device driver'. The location of the sysfs files containing the masks are:

```
/sys/bus/ap/apmask
/sys/bus/ap/aqmask
```

   The 'apmask' is a 256-bit mask that identifies a set of AP adapter IDs (APID). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APID from 0-255. If a bit is set, the APID is marked as usable only by the default AP queue device drivers; otherwise, the APID is usable by the vfio_ap device driver.

   The 'aqmask' is a 256-bit mask that identifies a set of AP queue indexes (APQI). Each bit in the mask, from left to right (i.e., from most significant to least significant bit in big endian order), corresponds to an APQI from 0-255. If a bit is set, the APQI is marked as usable only by the default AP queue device drivers; otherwise, the APQI is usable by the vfio_ap device driver.

   Take, for example, the following mask:

```
  0x7dffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

It indicates:

  1, 2, 3, 4, 5, and 7-255 belong to the default drivers' pool, and 0␣
↪and 6
  belong to the vfio_ap device driver's pool.
```

   The APQN of each AP queue device assigned to the linux host is checked by the AP bus against the set of APQNs derived from the cross product of APIDs and APQIs marked as usable only by the default AP queue device drivers. If a match is detected, only the default AP queue device drivers will be probed; otherwise, the vfio_ap device driver will be probed.

   By default, the two masks are set to reserve all APQNs for use by the default AP queue device drivers. There are two ways the default masks can be changed:

   1. The sysfs mask files can be edited by echoing a string into the respective sysfs mask file in one of two formats:

      - An absolute hex string starting with 0x - like "0x12345678" - sets the mask. If the given string is shorter than the mask, it is padded with 0s on the right; for example, specifying a mask value of 0x41 is the same as specifying:

```
0x4100000000000000000000000000000000000000000000000000000000000000
```

Keep in mind that the mask reads from left to right (i.e., most significant to least significant bit in big endian order), so the mask above identifies device numbers 1 and 7 (01000001).

If the string is longer than the mask, the operation is terminated with an error (EINVAL).

- Individual bits in the mask can be switched on and off by specifying each bit number to be switched in a comma separated list. Each bit number string must be prepended with a (‘+’) or minus (‘-‘) to indicate the corresponding bit is to be switched on (‘+’) or off (‘-‘). Some valid values are:
  - “+0” switches bit 0 on
  - “-13” switches bit 13 off
  - “+0x41” switches bit 65 on
  - “-0xff” switches bit 255 off

  The following example:

  ```
  +0,-6,+0x47,-0xf0
  ```

  Switches bits 0 and 71 (0x47) on

  Switches bits 6 and 240 (0xf0) off

  Note that the bits not specified in the list remain as they were before the operation.

2. The masks can also be changed at boot time via parameters on the kernel command line like this:

   ```
   ap.apmask=0xffff ap.aqmask=0x40
   ```

   This would create the following masks:

   ```
   apmask:
   0xffff0000000000000000000000000000000000000000000000000000000000000000

   aqmask:
   0x4000000000000000000000000000000000000000000000000000000000000000
   ```

   Resulting in these two pools:

   ```
   default drivers pool:    adapter 0-15, domain 1
   alternate drivers pool:  adapter 16-255, domains 0, 2-255
   ```

7.5.4 Securing the APQNs for our example

To secure the AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff for use by the vfio_ap device driver, the corresponding APQNs can either be removed from the default masks:

```
echo -5,-6 > /sys/bus/ap/apmask

echo -4,-0x47,-0xab,-0xff > /sys/bus/ap/aqmask
```

Or the masks can be set as follows:

```
echo␣
↪0xf9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff␣
↪\
> apmask

echo␣
↪0xf7fffffffffffffeffffffffffffffffffffffeffffffffffffffffffffffffe␣
↪\
> aqmask
```

This will result in AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff getting bound to the vfio_ap device driver. The sysfs directory for the vfio_ap device driver will now contain symbolic links to the AP queue devices bound to it:

```
/sys/bus/ap
... [drivers]
...... [vfio_ap]
......... [05.0004]
......... [05.0047]
......... [05.00ab]
......... [05.00ff]
......... [06.0004]
......... [06.0047]
......... [06.00ab]
......... [06.00ff]
```

Keep in mind that only type 10 and newer adapters (i.e., CEX4 and later) can be bound to the vfio_ap device driver. The reason for this is to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future and for which there are few older systems on which to test.

The administrator, therefore, must take care to secure only AP queues that can be bound to the vfio_ap device driver. The device type for a given AP queue device can be read from the parent card’s sysfs directory. For example, to see the hardware type of the queue 05.0004:

```
cat /sys/bus/ap/devices/card05/hwtype
```

The hwtype must be 10 or higher (CEX4 or newer) in order to be bound to the vfio_ap device driver.

3. Create the mediated devices needed to configure the AP matrixes for the three guests and to provide an interface to the vfio_ap driver for use by the

guests:

```
/sys/devices/vfio_ap/matrix/
--- [mdev_supported_types]
------ [vfio_ap-passthrough] (passthrough mediated matrix device type)
--------- create
--------- [devices]
```

To create the mediated devices for the three guests:

```
uuidgen > create
uuidgen > create
uuidgen > create

or

echo $uuid1 > create
echo $uuid2 > create
echo $uuid3 > create
```

This will create three mediated devices in the [devices] subdirectory named after the UUID written to the create attribute file. We call them $uuid1, $uuid2 and $uuid3 and this is the sysfs directory structure after creation:

```
/sys/devices/vfio_ap/matrix/
--- [mdev_supported_types]
------ [vfio_ap-passthrough]
--------- [devices]
------------ [$uuid1]
---------------- assign_adapter
---------------- assign_control_domain
---------------- assign_domain
---------------- matrix
---------------- unassign_adapter
---------------- unassign_control_domain
---------------- unassign_domain

------------ [$uuid2]
---------------- assign_adapter
---------------- assign_control_domain
---------------- assign_domain
---------------- matrix
---------------- unassign_adapter
-----------------unassign_control_domain
-----------------unassign_domain

------------ [$uuid3]
---------------- assign_adapter
---------------- assign_control_domain
---------------- assign_domain
---------------- matrix
---------------- unassign_adapter
-----------------unassign_control_domain
-----------------unassign_domain
```

4. The administrator now needs to configure the matrixes for the mediated devices $uuid1 (for Guest1), $uuid2 (for Guest2) and $uuid3 (for Guest3).

This is how the matrix is configured for Guest1:

```
echo 5 > assign_adapter
echo 6 > assign_adapter
echo 4 > assign_domain
echo 0xab > assign_domain
```

Control domains can similarly be assigned using the assign_control_domain sysfs file.

If a mistake is made configuring an adapter, domain or control domain, you can use the unassign_xxx files to unassign the adapter, domain or control domain.

To display the matrix configuration for Guest1:

```
cat matrix
```

This is how the matrix is configured for Guest2:

```
echo 5 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

This is how the matrix is configured for Guest3:

```
echo 6 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

In order to successfully assign an adapter:

- The adapter number specified must represent a value from 0 up to the maximum adapter number configured for the system. If an adapter number higher than the maximum is specified, the operation will terminate with an error (ENODEV).
- All APQNs that can be derived from the adapter ID and the IDs of the previously assigned domains must be bound to the vfio_ap device driver. If no domains have yet been assigned, then there must be at least one APQN with the specified APID bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).

  No APQN that can be derived from the adapter ID and the IDs of the previously assigned domains can be assigned to another mediated matrix device. If an APQN is assigned to another mediated matrix device, the operation will terminate with an error (EADDRINUSE).

In order to successfully assign a domain:

- The domain number specified must represent a value from 0 up to the maximum domain number configured for the system. If a domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).
- All APQNs that can be derived from the domain ID and the IDs of the previously assigned adapters must be bound to the vfio_ap device driver.

If no domains have yet been assigned, then there must be at least one APQN with the specified APQI bound to the vfio_ap driver. If no such APQNs are bound to the driver, the operation will terminate with an error (EADDRNOTAVAIL).

No APQN that can be derived from the domain ID and the IDs of the previously assigned adapters can be assigned to another mediated matrix device. If an APQN is assigned to another mediated matrix device, the operation will terminate with an error (EADDRINUSE).

In order to successfully assign a control domain, the domain number specified must represent a value from 0 up to the maximum domain number configured for the system. If a control domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

5. Start Guest1:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid1 ...
```

7. Start Guest2:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid2 ...
```

7. Start Guest3:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid3 ...
```

When the guest is shut down, the mediated matrix devices may be removed.

Using our example again, to remove the mediated matrix device $uuid1:

```
/sys/devices/vfio_ap/matrix/
   --- [mdev_supported_types]
   ------ [vfio_ap-passthrough]
   --------- [devices]
   ------------ [$uuid1]
   --------------- remove
```

```
echo 1 > remove
```

This will remove all of the mdev matrix device's sysfs structures including the mdev device itself. To recreate and reconfigure the mdev matrix device, all of the steps starting with step 3 will have to be performed again. Note that the remove will fail if a guest using the mdev is still running.

It is not necessary to remove an mdev matrix device, but one may want to remove it if no guest will use it during the remaining lifetime of the linux host. If the mdev matrix device is removed, one may want to also reconfigure the pool of adapters and queues reserved for use by the default drivers.

7.6 Limitations

- The KVM/kernel interfaces do not provide a way to prevent restoring an APQN to the default drivers pool of a queue that is still assigned to a mediated device in use by a guest. It is incumbent upon the administrator to ensure there is no mediated device in use by a guest to which the APQN is assigned lest the host be given access to the private data of the AP queue device such as a private key configured specifically for the guest.
- Dynamically modifying the AP matrix for a running guest (which would amount to hot(un)plug of AP devices for the guest) is currently not supported
- Live guest migration is not supported for guests using AP devices.

CHAPTER

EIGHT

VFIO-CCW: THE BASIC INFRASTRUCTURE

8.1 Introduction

Here we describe the vfio support for I/O subchannel devices for Linux/s390. Motivation for vfio-ccw is to passthrough subchannels to a virtual machine, while vfio is the means.

Different than other hardware architectures, s390 has defined a unified I/O access method, which is so called Channel I/O. It has its own access patterns:

- Channel programs run asynchronously on a separate (co)processor.
- The channel subsystem will access any memory designated by the caller in the channel program directly, i.e. there is no iommu involved.

Thus when we introduce vfio support for these devices, we realize it with a mediated device (mdev) implementation. The vfio mdev will be added to an iommu group, so as to make itself able to be managed by the vfio framework. And we add read/write callbacks for special vfio I/O regions to pass the channel programs from the mdev to its parent device (the real I/O subchannel device) to do further address translation and to perform I/O instructions.

This document does not intend to explain the s390 I/O architecture in every detail. More information/reference could be found here:

- A good start to know Channel I/O in general: https://en.wikipedia.org/wiki/Channel_I/O
- s390 architecture: s390 Principles of Operation manual (IBM Form. No. SA22-7832)
- The existing QEMU code which implements a simple emulated channel subsystem could also be a good reference. It makes it easier to follow the flow. qemu/hw/s390x/css.c

For vfio mediated device framework: - Documentation/driver-api/vfio-mediated-device.rst

8.2 Motivation of vfio-ccw

Typically, a guest virtualized via QEMU/KVM on s390 only sees paravirtualized virtio devices via the “Virtio Over Channel I/O (virtio-ccw)” transport. This makes virtio devices discoverable via standard operating system algorithms for handling channel devices.

However this is not enough. On s390 for the majority of devices, which use the standard Channel I/O based mechanism, we also need to provide the functionality of passing through them to a QEMU virtual machine. This includes devices that don’t have a virtio counterpart (e.g. tape drives) or that have specific characteristics which guests want to exploit.

For passing a device to a guest, we want to use the same interface as everybody else, namely vfio. We implement this vfio support for channel devices via the vfio mediated device framework and the subchannel device driver “vfio_ccw”.

8.3 Access patterns of CCW devices

s390 architecture has implemented a so called channel subsystem, that provides a unified view of the devices physically attached to the systems. Though the s390 hardware platform knows about a huge variety of different peripheral attachments like disk devices (aka. DASDs), tapes, communication controllers, etc. They can all be accessed by a well defined access method and they are presenting I/O completion a unified way: I/O interruptions.

All I/O requires the use of channel command words (CCWs). A CCW is an instruction to a specialized I/O channel processor. A channel program is a sequence of CCWs which are executed by the I/O channel subsystem. To issue a channel program to the channel subsystem, it is required to build an operation request block (ORB), which can be used to point out the format of the CCW and other control information to the system. The operating system signals the I/O channel subsystem to begin executing the channel program with a SSCH (start sub-channel) instruction. The central processor is then free to proceed with non-I/O instructions until interrupted. The I/O completion result is received by the interrupt handler in the form of interrupt response block (IRB).

Back to vfio-ccw, in short:

- ORBs and channel programs are built in guest kernel (with guest physical addresses).
- ORBs and channel programs are passed to the host kernel.
- Host kernel translates the guest physical addresses to real addresses and starts the I/O with issuing a privileged Channel I/O instruction (e.g SSCH).
- channel programs run asynchronously on a separate processor.
- I/O completion will be signaled to the host with I/O interruptions. And it will be copied as IRB to user space to pass it back to the guest.

8.4 Physical vfio ccw device and its child mdev

As mentioned above, we realize vfio-ccw with a mdev implementation.

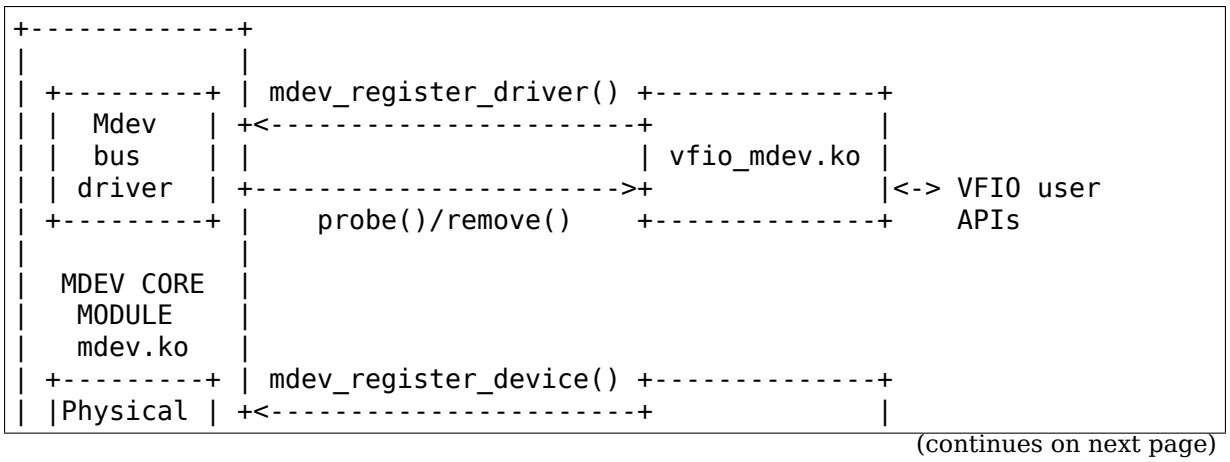
Channel I/O does not have IOMMU hardware support, so the physical vfio-ccw device does not have an IOMMU level translation or isolation.

Subchannel I/O instructions are all privileged instructions. When handling the I/O instruction interception, vfio-ccw has the software policing and translation how the channel program is programmed before it gets sent to hardware.

Within this implementation, we have two drivers for two types of devices:

- The vfio_ccw driver for the physical subchannel device. This is an I/O subchannel driver for the real subchannel device. It realizes a group of callbacks and registers to the mdev framework as a parent (physical) device. As a consequence, mdev provides vfio_ccw a generic interface (sysfs) to create mdev devices. A vfio mdev could be created by vfio_ccw then and added to the mediated bus. It is the vfio device that added to an IOMMU group and a vfio group. vfio_ccw also provides an I/O region to accept channel program request from user space and store I/O interrupt result for user space to retrieve. To notify user space an I/O completion, it offers an interface to setup an eventfd fd for asynchronous signaling.
- The vfio_mdev driver for the mediated vfio ccw device. This is provided by the mdev framework. It is a vfio device driver for the mdev that created by vfio_ccw. It realizes a group of vfio device driver callbacks, adds itself to a vfio group, and registers itself to the mdev framework as a mdev driver. It uses a vfio iommu backend that uses the existing map and unmap ioctls, but rather than programming them into an IOMMU for a device, it simply stores the translations for use by later requests. This means that a device programmed in a VM with guest physical addresses can have the vfio kernel convert that address to process virtual address, pin the page and program the hardware with the host physical address in one step. For a mdev, the vfio iommu backend will not pin the pages during the VFIO_IOMMU_MAP_DMA ioctl. Mdev framework will only maintain a database of the iova<->vaddr mappings in this operation. And they export a vfio_pin_pages and a vfio_unpin_pages interfaces from the vfio iommu backend for the physical devices to pin and unpin pages by demand.

Below is a high Level block diagram:



(continues on next page)

(continued from previous page)

```
| | device  | |                        |  vfio_ccw.ko |<-> subchannel
| |interface| +----------------------->+              |     device
| +---------+ |       callback         +--------------+
+-------------+
```

The process of how these work together.

1. vfio_ccw.ko drives the physical I/O subchannel, and registers the physical device (with callbacks) to mdev framework. When vfio_ccw probing the subchannel device, it registers device pointer and callbacks to the mdev framework. Mdev related file nodes under the device node in sysfs would be created for the subchannel device, namely 'mdev_create', 'mdev_destroy' and 'mdev_supported_types'.
2. Create a mediated vfio ccw device. Use the 'mdev_create' sysfs file, we need to manually create one (and only one for our case) mediated device.
3. vfio_mdev.ko drives the mediated ccw device. vfio_mdev is also the vfio device drvier. It will probe the mdev and add it to an iommu_group and a vfio_group. Then we could pass through the mdev to a guest.

8.5 VFIO-CCW Regions

The vfio-ccw driver exposes MMIO regions to accept requests from and return results to userspace.

8.6 vfio-ccw I/O region

An I/O region is used to accept channel program request from user space and store I/O interrupt result for user space to retrieve. The definition of the region is:

```
struct ccw_io_region {
#define ORB_AREA_SIZE 12
        __u8    orb_area[ORB_AREA_SIZE];
#define SCSW_AREA_SIZE 12
        __u8    scsw_area[SCSW_AREA_SIZE];
#define IRB_AREA_SIZE 96
        __u8    irb_area[IRB_AREA_SIZE];
        __u32   ret_code;
} __packed;
```

This region is always available.

While starting an I/O request, orb_area should be filled with the guest ORB, and scsw_area should be filled with the SCSW of the Virtual Subchannel.

irb_area stores the I/O result.

ret_code stores a return code for each access of the region. The following values may occur:

0 The operation was successful.

-EOPNOTSUPP The orb specified transport mode or an unidentified IDAW format, or the scsw specified a function other than the start function.

-EIO A request was issued while the device was not in a state ready to accept requests, or an internal error occurred.

-EBUSY The subchannel was status pending or busy, or a request is already active.

-EAGAIN A request was being processed, and the caller should retry.

-EACCES The channel path(s) used for the I/O were found to be not operational.

-ENODEV The device was found to be not operational.

-EINVAL The orb specified a chain longer than 255 ccws, or an internal error occurred.

8.7 vfio-ccw cmd region

The vfio-ccw cmd region is used to accept asynchronous instructions from userspace:

```
#define VFIO_CCW_ASYNC_CMD_HSCH (1 << 0)
#define VFIO_CCW_ASYNC_CMD_CSCH (1 << 1)
struct ccw_cmd_region {
       __u32 command;
       __u32 ret_code;
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_ASYNC_CMD.

Currently, CLEAR SUBCHANNEL and HALT SUBCHANNEL use this region.

command specifies the command to be issued; ret_code stores a return code for each access of the region. The following values may occur:

0 The operation was successful.

-ENODEV The device was found to be not operational.

-EINVAL A command other than halt or clear was specified.

-EIO A request was issued while the device was not in a state ready to accept requests.

-EAGAIN A request was being processed, and the caller should retry.

-EBUSY The subchannel was status pending or busy while processing a halt request.

8.8 vfio-ccw schib region

The vfio-ccw schib region is used to return Subchannel-Information Block (SCHIB) data to userspace:

```
struct ccw_schib_region {
#define SCHIB_AREA_SIZE 52
        __u8 schib_area[SCHIB_AREA_SIZE];
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_SCHIB.

Reading this region triggers a STORE SUBCHANNEL to be issued to the associated hardware.

8.9 vfio-ccw crw region

The vfio-ccw crw region is used to return Channel Report Word (CRW) data to userspace:

```
struct ccw_crw_region {
        __u32 crw;
        __u32 pad;
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_CRW.

Reading this region returns a CRW if one that is relevant for this subchannel (e.g. one reporting changes in channel path state) is pending, or all zeroes if not. If multiple CRWs are pending (including possibly chained CRWs), reading this region again will return the next one, until no more CRWs are pending and zeroes are returned. This is similar to how STORE CHANNEL REPORT WORD works.

8.10 vfio-ccw operation details

vfio-ccw follows what vfio-pci did on the s390 platform and uses vfio-iommu-type1 as the vfio iommu backend.

- CCW translation APIs A group of APIs (start with cp_) to do CCW translation. The CCWs passed in by a user space program are organized with their guest physical memory addresses. These APIs will copy the CCWs into kernel space, and assemble a runnable kernel channel program by updating the guest physical addresses with their corresponding host physical addresses. Note that we have to use IDALs even for direct-access CCWs, as the referenced memory can be located anywhere, including above 2G.
- vfio_ccw device driver This driver utilizes the CCW translation APIs and introduces vfio_ccw, which is the driver for the I/O subchannel devices you want to pass through. vfio_ccw implements the following vfio ioctls:

```
VFIO_DEVICE_GET_INFO
VFIO_DEVICE_GET_IRQ_INFO
VFIO_DEVICE_GET_REGION_INFO
VFIO_DEVICE_RESET
VFIO_DEVICE_SET_IRQS
```

This provides an I/O region, so that the user space program can pass a channel program to the kernel, to do further CCW translation before issuing them to a real device. This also provides the SET_IRQ ioctl to setup an event notifier to notify the user space program the I/O completion in an asynchronous way.

The use of vfio-ccw is not limited to QEMU, while QEMU is definitely a good example to get understand how these patches work. Here is a little bit more detail how an I/O request triggered by the QEMU guest will be handled (without error handling).

Explanation:

- Q1-Q7: QEMU side process.
- K1-K5: Kernel side process.

Q1. Get I/O region info during initialization.

Q2. Setup event notifier and handler to handle I/O completion.

……

Q3. Intercept a ssch instruction.

Q4. Write the guest channel program and ORB to the I/O region.

K1. Copy from guest to kernel.

K2. Translate the guest channel program to a host kernel space channel program, which becomes runnable for a real device.

K3. With the necessary information contained in the orb passed in by QEMU, issue the ccwchain to the device.

K4. Return the ssch CC code.

Q5. Return the CC code to the guest.

……

K5. Interrupt handler gets the I/O result and write the result to the I/O region.

K6. Signal QEMU to retrieve the result.

Q6. Get the signal and event handler reads out the result from the I/O region.

Q7. Update the irb for the guest.

8.11 Limitations

The current vfio-ccw implementation focuses on supporting basic commands needed to implement block device functionality (read/write) of DASD/ECKD device only. Some commands may need special handling in the future, for example, anything related to path grouping.

DASD is a kind of storage device. While ECKD is a data recording format. More information for DASD and ECKD could be found here: https://en.wikipedia.org/wiki/Direct-access_storage_device https://en.wikipedia.org/wiki/Count_key_data

Together with the corresponding work in QEMU, we can bring the passed through DASD/ECKD device online in a guest now and use it as a block device.

The current code allows the guest to start channel programs via START SUBCHANNEL, and to issue HALT SUBCHANNEL, CLEAR SUBCHANNEL, and STORE SUBCHANNEL.

Currently all channel programs are prefetched, regardless of the p-bit setting in the ORB. As a result, self modifying channel programs are not supported. For this reason, IPL has to be handled as a special case by a userspace/guest program; this has been implemented in QEMU's s390-ccw bios as of QEMU 4.1.

vfio-ccw supports classic (command mode) channel I/O only. Transport mode (HPF) is not supported.

QDIO subchannels are currently not supported. Classic devices other than DASD/ECKD might work, but have not been tested.

8.12 Reference

1. ESA/s390 Principles of Operation manual (IBM Form. No. SA22-7832)
2. ESA/390 Common I/O Device Commands manual (IBM Form. No. SA22-7204)
3. https://en.wikipedia.org/wiki/Channel_I/O
4. Documentation/s390/cds.rst
5. Documentation/driver-api/vfio.rst
6. Documentation/driver-api/vfio-mediated-device.rst

CHAPTER

NINE

THE S390 SCSI DUMP TOOL (ZFCPDUMP)

System z machines (z900 or higher) provide hardware support for creating system dumps on SCSI disks. The dump process is initiated by booting a dump tool, which has to create a dump of the current (probably crashed) Linux image. In order to not overwrite memory of the crashed Linux with data of the dump tool, the hardware saves some memory plus the register sets of the boot CPU before the dump tool is loaded. There exists an SCLP hardware interface to obtain the saved memory afterwards. Currently 32 MB are saved.

This zfcpdump implementation consists of a Linux dump kernel together with a user space dump tool, which are loaded together into the saved memory region below 32 MB. zfcpdump is installed on a SCSI disk using zipl (as contained in the s390-tools package) to make the device bootable. The operator of a Linux system can then trigger a SCSI dump by booting the SCSI disk, where zfcpdump resides on.

The user space dump tool accesses the memory of the crashed system by means of the /proc/vmcore interface. This interface exports the crashed system's memory and registers in ELF core dump format. To access the memory which has been saved by the hardware SCLP requests will be created at the time the data is needed by /proc/vmcore. The tail part of the crashed systems memory which has not been stashed by hardware can just be copied from real memory.

To build a dump enabled kernel the kernel config option CONFIG_CRASH_DUMP has to be set.

To get a valid zfcpdump kernel configuration use "make zfcpdump_defconfig".

The s390 zipl tool looks for the zfcpdump kernel and optional initrd/initramfs under the following locations:

- kernel: <zfcpdump directory>/zfcpdump.image
- ramdisk: <zfcpdump directory>/zfcpdump.rd

The zfcpdump directory is defined in the s390-tools package.

The user space application of zfcpdump can reside in an intitramfs or an initrd. It can also be included in a built-in kernel initramfs. The application reads from /proc/vmcore or zcore/mem and writes the system dump to a SCSI disk.

The s390-tools package version 1.24.0 and above builds an external zfcpdump initramfs with a user space application that writes the dump to a SCSI partition.

For more information on how to use zfcpdump refer to the s390 'Using the Dump Tools' book, which is available from IBM Knowledge Center: https://www.ibm.com/

support/knowledgecenter/linuxonibm/liaaf/lnz_r_dt.html

CHAPTER

TEN

S/390 COMMON I/O-LAYER

10.1 command line parameters, procfs and debugfs entries

10.1.1 Command line parameters

- ccw_timeout_log

  Enable logging of debug information in case of ccw device timeouts.

- cio_ignore = device[,device[,..]]

  device := {all | [!]ipldev | [!]condev | [!]<devno> | [!]<devno>-<devno>}

  The given devices will be ignored by the common I/O-layer; no detection and device sensing will be done on any of those devices. The subchannel to which the device in question is attached will be treated as if no device was attached.

  An ignored device can be un-ignored later; see the "/proc entries" -section for details.

  The devices must be given either as bus ids (0.x.abcd) or as hexadecimal device numbers (0xabcd or abcd, for 2.4 backward compatibility). If you give a device number 0xabcd, it will be interpreted as 0.0.abcd.

  You can use the 'all' keyword to ignore all devices. The 'ipldev' and 'condev' keywords can be used to refer to the CCW based boot device and CCW console device respectively (these are probably useful only when combined with the '!' operator). The '!' operator will cause the I/O-layer to _not_ ignore a device. The command line is parsed from left to right.

  For example:

```
cio_ignore=0.0.0023-0.0.0042,0.0.4711
```

  will ignore all devices ranging from 0.0.0023 to 0.0.0042 and the device 0.0.4711, if detected.

  As another example:

```
cio_ignore=all,!0.0.4711,!0.0.fd00-0.0.fd02
```

  will ignore all devices but 0.0.4711, 0.0.fd00, 0.0.fd01, 0.0.fd02.

By default, no devices are ignored.

10.1.2 /proc entries

- /proc/cio_ignore

  Lists the ranges of devices (by bus id) which are ignored by common I/O.

  You can un-ignore certain or all devices by piping to /proc/cio_ignore. "free all" will un-ignore all ignored devices, "free <device range>, <device range>, ⋯" will un-ignore the specified devices.

  For example, if devices 0.0.0023 to 0.0.0042 and 0.0.4711 are ignored,

  - echo free 0.0.0030-0.0.0032 > /proc/cio_ignore will un-ignore devices 0.0.0030 to 0.0.0032 and will leave devices 0.0.0023 to 0.0.002f, 0.0.0033 to 0.0.0042 and 0.0.4711 ignored;
  - echo free 0.0.0041 > /proc/cio_ignore will furthermore un-ignore device 0.0.0041;
  - echo free all > /proc/cio_ignore will un-ignore all remaining ignored devices.

  When a device is un-ignored, device recognition and sensing is performed and the device driver will be notified if possible, so the device will become available to the system. Note that un-ignoring is performed asynchronously.

  You can also add ranges of devices to be ignored by piping to /proc/cio_ignore; "add <device range>, <device range>, ⋯" will ignore the specified devices.

  Note: While already known devices can be added to the list of devices to be ignored, there will be no effect on then. However, if such a device disappears and then reappears, it will then be ignored. To make known devices go away, you need the "purge" command (see below).

  For example:

```
"echo add 0.0.a000-0.0.accc, 0.0.af00-0.0.afff > /proc/cio_ignore"
```

  will add 0.0.a000-0.0.accc and 0.0.af00-0.0.afff to the list of ignored devices.

  You can remove already known but now ignored devices via:

```
"echo purge > /proc/cio_ignore"
```

  All devices ignored but still registered and not online (= not in use) will be deregistered and thus removed from the system.

  The devices can be specified either by bus id (0.x.abcd) or, for 2.4 backward compatibility, by the device number in hexadecimal (0xabcd or abcd). Device numbers given as 0xabcd will be interpreted as 0.0.abcd.

- /proc/cio_settle

  A write request to this file is blocked until all queued cio actions are handled. This will allow userspace to wait for pending work affecting device availability after changing cio_ignore or the hardware configuration.

- For some of the information present in the /proc filesystem in 2.4 (namely, /proc/subchannels and /proc/chpids), see driver-model.txt. Information formerly in /proc/irq_count is now in /proc/interrupts.

10.1.3 debugfs entries

- /sys/kernel/debug/s390dbf/cio_*/ (S/390 debug feature)

  Some views generated by the debug feature to hold various debug outputs.

  - /sys/kernel/debug/s390dbf/cio_crw/sprintf Messages from the processing of pending channel report words (machine check handling).
  - /sys/kernel/debug/s390dbf/cio_msg/sprintf Various debug messages from the common I/O-layer.
  - /sys/kernel/debug/s390dbf/cio_trace/hex_ascii Logs the calling of functions in the common I/O-layer and, if applicable, which subchannel they were called for, as well as dumps of some data structures (like irb in an error case).

  The level of logging can be changed to be more or less verbose by piping to /sys/kernel/debug/s390dbf/cio_*/level a number between 0 and 6; see the documentation on the S/390 debug feature (Documentation/s390/s390dbf.rst) for details.

CHAPTER

ELEVEN

S/390 PCI

Authors:

- Pierre Morel

11.1 Command line parameters and debugfs entries

11.1.1 Command line parameters

- nomio

  Do not use PCI Mapped I/O (MIO) instructions.

- norid

  Ignore the RID field and force use of one PCI domain per PCI function.

11.1.2 debugfs entries

The S/390 debug feature (s390dbf) generates views to hold various debug results in sysfs directories of the form:

- /sys/kernel/debug/s390dbf/pci_*/

For example:

- /sys/kernel/debug/s390dbf/pci_msg/sprintf Holds messages from the processing of PCI events, like machine check handling and setting of global functionality, like UID checking.

Change the level of logging to be more or less verbose by piping a number between 0 and 6 to /sys/kernel/debug/s390dbf/pci_*/level. For details, see the documentation on the S/390 debug feature at Documentation/s390/s390dbf.rst.

11.2 Sysfs entries

Entries specific to zPCI functions and entries that hold zPCI information.

- /sys/bus/pci/slots/XXXXXXXX

  The slot entries are set up using the function identifier (FID) of the PCI function.

  - /sys/bus/pci/slots/XXXXXXXX/power

  A physical function that currently supports a virtual function cannot be powered off until all virtual functions are removed with: echo 0 > /sys/bus/pci/devices/XXXX:XX:XX.X/sriov_numvf

- /sys/bus/pci/devices/XXXX:XX:XX.X/

  - function_id A zPCI function identifier that uniquely identifies the function in the Z server.
  - function_handle Low-level identifier used for a configured PCI function. It might be useful for debuging.
  - pchid Model-dependent location of the I/O adapter.
  - pfgid PCI function group ID, functions that share identical functionality use a common identifier. A PCI group defines interrupts, IOMMU, IOTLB, and DMA specifics.
  - vfn The virtual function number, from 1 to N for virtual functions, 0 for physical functions.
  - pft The PCI function type
  - port The port corresponds to the physical port the function is attached to. It also gives an indication of the physical function a virtual function is attached to.
  - uid The unique identifier (UID) is defined when configuring an LPAR and is unique in the LPAR.
  - pfip/segmentX The segments determine the isolation of a function. They correspond to the physical path to the function. The more the segments are different, the more the functions are isolated.

11.3 Enumeration and hotplug

The PCI address consists of four parts: domain, bus, device and function, and is of this form: DDDD:BB:dd.f

- When not using multi-functions (norid is set, or the firmware does not support multi-functions):

  - There is only one function per domain.
  - The domain is set from the zPCI function’s UID as defined during the LPAR creation.

- When using multi-functions (norid parameter is not set), zPCI functions are addressed differently:
  - There is still only one bus per domain.
  - There can be up to 256 functions per bus.
  - The domain part of the address of all functions for a multi-Function device is set from the zPCI function's UID as defined in the LPAR creation for the function zero.
  - New functions will only be ready for use after the function zero (the function with devfn 0) has been enumerated.

CHAPTER

TWELVE

IBM 3270 CHANGELOG

```
ChangeLog for the UTS Global 3270-support patch

Sep 2002:       Get bootup colors right on 3270 console
        * In tubttybld.c, substantially revise ESC processing so␣
↪that
          ESC sequences (especially coloring ones) and the strings
          they affect work as right as 3270 can get them.  Also, set
          screen height to omit the two rows used for input area, in
          tty3270_open() in tubtty.c.

Sep 2002:       Dynamically get 3270 input buffer
        * Oversize 3270 screen widths may exceed GEOM_MAXINPLEN␣
↪columns,
          so get input-area buffer dynamically when sizing the␣
↪device in
          tubmakemin() in tuball.c (if it's the console) or tty3270_
↪open()
          in tubtty.c (if needed).  Change tubp->tty_input to be a
          pointer rather than an array, in tubio.h.

Sep 2002:       Fix tubfs kmalloc()s
        * Do read and write lengths correctly in fs3270_read()
          and fs3270_write(), while never asking kmalloc()
          for more than 0x800 bytes.  Affects tubfs.c and tubio.h.

Sep 2002:       Recognize 3270 control unit type 3174
        * Recognize control-unit type 0x3174 as well as 0x327?.
          The IBM 2047 device emulates a 3174 control unit.
          Modularize control-unit recognition in tuball.c by
          adding and invoking new tub3270_is_ours().

Apr 2002:       Fix 3270 console reboot loop
        * (Belated log entry) Fixed reboot loop if 3270 console,
          in tubtty.c:ttu3270_bh().

Feb 6, 2001:
        * This changelog is new
        * tub3270 now supports 3270 console:
                Specify y for CONFIG_3270 and y for CONFIG_3270_
```

```
↪CONSOLE.
                Support for 3215 will not appear if 3270 console␣
↪support
                is chosen.
                NOTE:  The default is 3270 console support, NOT␣
↪3215.
        * the components are remodularized: added source modules are
          tubttybld.c and tubttyscl.c, for screen-building code and
          scroll-timeout code.
        * tub3270 source for this (2.4.0) version is #ifdeffed to
          build with both 2.4.0 and 2.2.16.2.
        * color support and minimal other ESC-sequence support is␣
↪added.
```

CHAPTER

THIRTEEN

IBM 3270 CONFIG3270.SH

```
#!/bin/sh
#
# config3270 -- Autoconfigure /dev/3270/* and /etc/inittab
#
#       Usage:
#               config3270
#
#       Output:
#               /tmp/mkdev3270
#
#       Operation:
#               1. Run this script
#               2. Run the script it produces: /tmp/mkdev3270
#               3. Issue "telinit q" or reboot, as appropriate.
#
P=/proc/tty/driver/tty3270
ROOT=
D=$ROOT/dev
SUBD=3270
TTY=$SUBD/tty
TUB=$SUBD/tub
SCR=$ROOT/tmp/mkdev3270
SCRTMP=$SCR.a
GETTYLINE=:2345:respawn:/sbin/mingetty
INITTAB=$ROOT/etc/inittab
NINITTAB=$ROOT/etc/NEWinittab
OINITTAB=$ROOT/etc/OLDinittab
ADDNOTE=\\"# Additional mingettys for the 3270/tty* driver, tub3270 ---\\"

if ! ls $P > /dev/null 2>&1; then
        modprobe tub3270 > /dev/null 2>&1
fi
ls $P > /dev/null 2>&1 || exit 1

# Initialize two files, one for /dev/3270 commands and one
# to replace the /etc/inittab file (old one saved in OLDinittab)
echo "#!/bin/sh" > $SCR || exit 1
echo " " >> $SCR
echo "# Script built by /sbin/config3270" >> $SCR
if [ ! -d /dev/dasd ]; then
        echo rm -rf "$D/$SUBD/*" >> $SCR
fi
echo "grep -v $TTY $INITTAB > $NINITTAB" > $SCRTMP || exit 1
```

(continues on next page)

(continued from previous page)

```
echo "echo $ADDNOTE >> $NINITTAB" >> $SCRTMP
if [ ! -d /dev/dasd ]; then
        echo mkdir -p $D/$SUBD >> $SCR
fi

# Now query the tub3270 driver for 3270 device information
# and add appropriate mknod and mingetty lines to our files
echo what=config > $P
while read devno maj min;do
        if [ $min = 0 ]; then
                fsmaj=$maj
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TUB c $fsmaj 0 >> $SCR
                        echo chmod 666 $D/$TUB >> $SCR
                fi
        elif [ $maj = CONSOLE ]; then
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TUB$devno c $fsmaj $min >> $SCR
                fi
        else
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TTY$devno c $maj $min >>$SCR
                        echo mknod $D/$TUB$devno c $fsmaj $min >> $SCR
                fi
                echo "echo t$min$GETTYLINE $TTY$devno >> $NINITTAB" >>
↪$SCRTMP
        fi
done < $P

echo mv $INITTAB $OINITTAB >> $SCRTMP || exit 1
echo mv $NINITTAB $INITTAB >> $SCRTMP
cat $SCRTMP >> $SCR
rm $SCRTMP
exit 0
```