

# Writing s390 channel device drivers

Cornelia Huck <cornelia.huck@de.ibm.com>

---

# Writing s390 channel device drivers

by Cornelia Huck

Copyright © 2007 IBM Corp.

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

---

---

# Table of Contents

|  |    |
|--|----|
| 1. Introduction .....                            | 1  |
| 2. The ccw bus .....                             | 2  |
| I/O functions for channel-attached devices ..... | 2  |
| ccw devices .....                                | 16 |
| The channel-measurement facility .....           | 49 |
| 3. The ccwgroup bus .....                        | 55 |
| ccw group devices .....                          | 55 |
| 4. Generic interfaces .....                      | 65 |
| register_adapter_interrupt .....                 | 66 |
| unregister_adapter_interrupt .....               | 67 |
| airq_iv_create .....                             | 68 |
| airq_iv_release .....                            | 69 |
| airq_iv_alloc .....                              | 70 |
| airq_iv_free .....                               | 71 |
| airq_iv_scan .....                               | 72 |

---

# Chapter 1. Introduction

This document describes the interfaces available for device drivers that drive s390 based channel attached I/O devices. This includes interfaces for interaction with the hardware and interfaces for interacting with the common driver core. Those interfaces are provided by the s390 common I/O layer.

The document assumes a familiarity with the technical terms associated with the s390 channel I/O architecture. For a description of this architecture, please refer to the "z/Architecture: Principles of Operation", IBM publication no. SA22-7832.

While most I/O devices on a s390 system are typically driven through the channel I/O mechanism described here, there are various other methods (like the diag interface). These are out of the scope of this document.

Some additional information can also be found in the kernel source under Documentation/s390/driver-model.txt.

---

## Chapter 2. The ccw bus

The ccw bus typically contains the majority of devices available to a s390 system. Named after the channel command word (ccw), the basic command structure used to address its devices, the ccw bus contains so-called channel attached devices. They are addressed via I/O subchannels, visible on the css bus. A device driver for channel-attached devices, however, will never interact with the subchannel directly, but only via the I/O device on the ccw bus, the ccw device.

### I/O functions for channel-attached devices

Some hardware structures have been translated into C structures for use by the common I/O layer and device drivers. For more information on the hardware structures represented here, please consult the Principles of Operation.

## Name

struct ccw1 — channel command word

## Synopsis

```
struct ccw1 {  
    __u8 cmd_code;  
    __u8 flags;  
    __u16 count;  
    __u32 cda;  
};
```

## Members

|          |                                  |
|----------|----------------------------------|
| cmd_code | command code                     |
| flags    | flags, like IDA addressing, etc. |
| count    | byte count                       |
| cda      | data address                     |

## Description

The ccw is the basic structure to build channel programs that perform operations with the device or the control unit. Only Format-1 channel command words are supported.

## Name

struct erw — extended report word

## Synopsis

```
struct erw {
    __u32 res0:3;
    __u32 auth:1;
    __u32 pvrf:1;
    __u32 cpt:1;
    __u32 fsavf:1;
    __u32 cons:1;
    __u32 scavf:1;
    __u32 fsaf:1;
    __u32 scnt:6;
    __u32 res16:16;
};
```

## Members

|       |                                       |
|-------|---------------------------------------|
| res0  | reserved                              |
| auth  | authorization check                   |
| pvrf  | path-verification-required flag       |
| cpt   | channel-path timeout                  |
| fsavf | failing storage address validity flag |
| cons  | concurrent sense                      |
| scavf | secondary ccw address validity flag   |
| fsaf  | failing storage address format        |
| scnt  | sense count, if <i>cons</i> == 1      |
| res16 | reserved                              |

## Name

struct erw\_eadm — EADM Subchannel extended report word

## Synopsis

```
struct erw_eadm {  
    __u32 b:1;  
    __u32 r:1;  
};
```

## Members

b    aob error

r    arsb error



## Name

struct sublog — subchannel logout area

## Synopsis

```
struct sublog {
    __u32 res0:1;
    __u32 esf:7;
    __u32 lpum:8;
    __u32 arep:1;
    __u32 fvf:5;
    __u32 sacc:2;
    __u32 termc:2;
    __u32 devsc:1;
    __u32 serr:1;
    __u32 ioerr:1;
    __u32 seqc:3;
};
```

## Members

|       |                       |
|-------|-----------------------|
| res0  | reserved              |
| esf   | extended status flags |
| lpum  | last path used mask   |
| arep  | ancillary report      |
| fvf   | field-validity flags  |
| sacc  | storage access code   |
| termc | termination code      |
| devsc | device-status check   |
| serr  | secondary error       |
| ioerr | i/o-error alert       |
| seqc  | sequence code         |

## Name

struct esw0 — Format 0 Extended Status Word (ESW)

## Synopsis

```
struct esw0 {  
    struct sublog sublog;  
    struct erw erw;  
    __u32 faddr[2];  
    __u32 saddr;  
};
```

## Members

|          |                         |
|----------|-------------------------|
| sublog   | subchannel logout       |
| erw      | extended report word    |
| faddr[2] | failing storage address |
| saddr    | secondary ccw address   |

## Name

struct esw1 — Format 1 Extended Status Word (ESW)

## Synopsis

```
struct esw1 {  
    __u8 zero0;  
    __u8 lpum;  
    __u16 zero16;  
    struct erw erw;  
    __u32 zeros[3];  
};
```

## Members

|          |                          |
|----------|--------------------------|
| zero0    | reserved zeros           |
| lpum     | last path used mask      |
| zero16   | reserved zeros           |
| erw      | extended report word     |
| zeros[3] | three fullwords of zeros |

## Name

struct esw2 — Format 2 Extended Status Word (ESW)

## Synopsis

```
struct esw2 {  
    __u8 zero0;  
    __u8 lpum;  
    __u16 dcti;  
    struct erw erw;  
    __u32 zeros[3];  
};
```

## Members

|          |                              |
|----------|------------------------------|
| zero0    | reserved zeros               |
| lpum     | last path used mask          |
| dcti     | device-connect-time interval |
| erw      | extended report word         |
| zeros[3] | three fullwords of zeros     |

## Name

struct esw3 — Format 3 Extended Status Word (ESW)

## Synopsis

```
struct esw3 {  
    __u8 zero0;  
    __u8 lpum;  
    __u16 res;  
    struct erw erw;  
    __u32 zeros[3];  
};
```

## Members

|          |                          |
|----------|--------------------------|
| zero0    | reserved zeros           |
| lpum     | last path used mask      |
| res      | reserved                 |
| erw      | extended report word     |
| zeros[3] | three fullwords of zeros |

## Name

struct esw\_eadm — EADM Subchannel Extended Status Word (ESW)

## Synopsis

```
struct esw_eadm {  
    __u32 sublog;  
    struct erw_eadm erw;  
};
```

## Members

|        |                      |
|--------|----------------------|
| sublog | subchannel logout    |
| erw    | extended report word |

## Name

struct irb — interruption response block

## Synopsis

```
struct irb {  
    union scsw scsw;  
    union esw;  
    __u8 ecw[32];  
};
```

## Members

|         |                        |
|---------|------------------------|
| scsw    | subchannel status word |
| esw     | extended status word   |
| ecw[32] | extended control word  |

## Description

The irb that is handed to the device driver when an interrupt occurs. For solicited interrupts, the common I/O layer already performs checks whether a field is valid; a field not being valid is always passed as 0. If a unit check occurred, *ecw* may contain sense data; this is retrieved by the common I/O layer itself if the device doesn't support concurrent sense (so that the device driver never needs to perform basic sense itself). For unsolicited interrupts, the irb is passed as-is (expect for sense data, if applicable).

## Name

struct ciw — command information word (CIW) layout

## Synopsis

```
struct ciw {  
    __u32 et:2;  
    __u32 reserved:2;  
    __u32 ct:4;  
    __u32 cmd:8;  
    __u32 count:16;  
};
```

## Members

|          |               |
|----------|---------------|
| et       | entry type    |
| reserved | reserved bits |
| ct       | command type  |
| cmd      | command code  |
| count    | command count |



## Name

`struct ccw_dev_id` — unique identifier for ccw devices

## Synopsis

```
struct ccw_dev_id {  
    u8 ssid;  
    u16 devno;  
};
```

## Members

|                    |                   |
|--------------------|-------------------|
| <code>ssid</code>  | subchannel set id |
| <code>devno</code> | device number     |

## Description

This structure is not directly based on any hardware structure. The hardware identifies a device by its device number and its subchannel, which is in turn identified by its id. In order to get a unique identifier for ccw devices across subchannel sets, *struct ccw\_dev\_id* has been introduced.

## Name

`ccw_dev_id_is_equal` — compare two `ccw_dev_ids`

## Synopsis

```
int ccw_dev_id_is_equal (struct ccw_dev_id * dev_id1, struct ccw_dev_id  
* dev_id2);
```

## Arguments

*dev\_id1*    a `ccw_dev_id`

*dev\_id2*    another `ccw_dev_id`

## Returns

1 if the two structures are equal field-by-field, 0 if not.

## Context

any

## Name

`pathmask_to_pos` — find the position of the left-most bit in a pathmask

## Synopsis

```
u8 pathmask_to_pos (u8 mask);
```

## Arguments

*mask* pathmask with at least one bit set

## ccw devices

Devices that want to initiate channel I/O need to attach to the ccw bus. Interaction with the driver core is done via the common I/O layer, which provides the abstractions of ccw devices and ccw device drivers.

The functions that initiate or terminate channel I/O all act upon a ccw device structure. Device drivers must not bypass those functions or strange side effects may happen.

## Name

struct ccw\_device — channel attached device

## Synopsis

```
struct ccw_device {
    spinlock_t * ccwlock;
    struct ccw_device_id id;
    struct ccw_driver * drv;
    struct device dev;
    int online;
    void (* handler) (struct ccw_device *, unsigned long, struct irb *);
};
```

## Members

|         |                            |
|---------|----------------------------|
| ccwlock | pointer to device lock     |
| id      | id of this device          |
| drv     | ccw driver for this device |
| dev     | embedded device structure  |
| online  | online status of device    |
| handler | interrupt handler          |

## Description

*handler* is a member of the device rather than the driver since a driver can have different interrupt handlers for different ccw devices (multi-subchannel drivers).

## Name

struct ccw\_driver — device driver for channel attached devices

## Synopsis

```
struct ccw_driver {
    struct ccw_device_id * ids;
    int (* probe) (struct ccw_device *);
    void (* remove) (struct ccw_device *);
    int (* set_online) (struct ccw_device *);
    int (* set_offline) (struct ccw_device *);
    int (* notify) (struct ccw_device *, int);
    void (* path_event) (struct ccw_device *, int *);
    void (* shutdown) (struct ccw_device *);
    int (* prepare) (struct ccw_device *);
    void (* complete) (struct ccw_device *);
    int (* freeze) (struct ccw_device *);
    int (* thaw) (struct ccw_device *);
    int (* restore) (struct ccw_device *);
    enum uc_todo (* uc_handler) (struct ccw_device *, struct irb *);
    struct device_driver driver;
    enum interruption_class int_class;
};
```

## Members

|             |   |
|-------------|---|
| ids         | ids supported by this driver                          |
| probe       | function called on probe                              |
| remove      | function called on remove                             |
| set_online  | called when setting device online                     |
| set_offline | called when setting device offline                    |
| notify      | notify driver of device state changes                 |
| path_event  | notify driver of channel path events                  |
| shutdown    | called at device shutdown                             |
| prepare     | prepare for pm state transition                       |
| complete    | undo work done in <i>prepare</i>                      |
| freeze      | callback for freezing during hibernation snapshotting |
| thaw        | undo work done in <i>freeze</i>                       |
| restore     | callback for restoring after hibernation              |
| uc_handler  | callback for unit check handler                       |
| driver      | embedded device driver structure                      |

int\_class      interruption class to use for accounting interrupts

## Name

`ccw_device_set_offline` — disable a ccw device for I/O

## Synopsis

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

## Arguments

*cdev*    target ccw device

## Description

This function calls the driver's `set_offline` function for *cdev*, if given, and then disables *cdev*.

## Returns

0 on success and a negative error value on failure.

## Context

enabled, ccw device lock not held

## Name

`ccw_device_set_online` — enable a ccw device for I/O

## Synopsis

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

## Arguments

*cdev*    target ccw device

## Description

This function first enables *cdev* and then calls the driver's `set_online` function for *cdev*, if given. If `set_online` returns an error, *cdev* is disabled again.

## Returns

0 on success and a negative error value on failure.

## Context

enabled, ccw device lock not held



## Name

`get_ccwdev_by_dev_id` — obtain device from a ccw device id

## Synopsis

```
struct ccw_device * get_ccwdev_by_dev_id (struct ccw_dev_id * dev_id);
```

## Arguments

*dev\_id* id of the device to be searched

## Description

This function searches all devices attached to the ccw bus for a device matching *dev\_id*.

## Returns

If a device is found its reference count is increased and returned; else `NULL` is returned.

## Name

`get_ccwdev_by_busid` — obtain device from a bus id

## Synopsis

```
struct ccw_device * get_ccwdev_by_busid (struct ccw_driver * cdrv, const  
char * bus_id);
```

## Arguments

*cdrv*      driver the device is owned by

*bus\_id*   bus id of the device to be searched

## Description

This function searches all devices owned by *cdrv* for a device with a bus id matching *bus\_id*.

## Returns

If a match is found, its reference count of the found device is increased and it is returned; else NULL is returned.

## Name

`ccw_driver_register` — register a ccw driver

## Synopsis

```
int ccw_driver_register (struct ccw_driver * cdriver);
```

## Arguments

*cdriver* driver to be registered

## Description

This function is mainly a wrapper around `driver_register`.

## Returns

0 on success and a negative error value on failure.

## Name

`ccw_driver_unregister` — deregister a ccw driver

## Synopsis

```
void ccw_driver_unregister (struct ccw_driver * cdriver);
```

## Arguments

*cdriver* driver to be deregistered

## Description

This function is mainly a wrapper around `driver_unregister`.

## Name

`ccw_device_siosl` — initiate logging

## Synopsis

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

## Arguments

*cdev*   ccw device

## Description

This function is used to invoke model-dependent logging within the channel subsystem.

## Name

`ccw_device_set_options_mask` — set some options and unset the rest

## Synopsis

```
int ccw_device_set_options_mask (struct ccw_device * cdev, unsigned long  
flags);
```

## Arguments

*cdev*    device for which the options are to be set

*flags*   options to be set

## Description

All flags specified in *flags* are set, all flags not specified in *flags* are cleared.

## Returns

0 on success, -EINVAL on an invalid flag combination.

## Name

`ccw_device_set_options` — set some options

## Synopsis

```
int ccw_device_set_options (struct ccw_device * cdev, unsigned long  
flags);
```

## Arguments

*cdev*    device for which the options are to be set

*flags*   options to be set

## Description

All flags specified in *flags* are set, the remainder is left untouched.

## Returns

0 on success, -EINVAL if an invalid flag combination would ensue.

## Name

`ccw_device_clear_options` — clear some options

## Synopsis

```
void ccw_device_clear_options (struct ccw_device * cdev, unsigned long  
flags);
```

## Arguments

*cdev*    device for which the options are to be cleared

*flags*   options to be cleared

## Description

All flags specified in *flags* are cleared, the remainder is left untouched.



## Name

`ccw_device_is_pathgroup` — determine if paths to this device are grouped

## Synopsis

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

## Arguments

*cdev*   ccw device

## Description

Return non-zero if there is a path group, zero otherwise.

## Name

`ccw_device_is_multipath` — determine if device is operating in multipath mode

## Synopsis

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

## Arguments

*cdev*   ccw device

## Description

Return non-zero if device is operating in multipath mode, zero otherwise.

## Name

`ccw_device_clear` — terminate I/O request processing

## Synopsis

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

## Arguments

*cdev*        target ccw device

*intparm*    interruption parameter; value is only used if no I/O is outstanding, otherwise the `intparm` associated with the I/O request is returned

## Description

`ccw_device_clear` calls `csch` on *cdev*'s subchannel.

## Returns

0 on success, `-ENODEV` on device not operational, `-EINVAL` on invalid device state.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_start_key` — start a s390 channel program with key

## Synopsis

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

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | target ccw device  |
| <i>cpa</i>     | logical start address of channel program   |
| <i>intparm</i> | user specific interruption parameter; will be presented back to <i>cdev</i> 's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request. |
| <i>lpm</i>     | defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.  |
| <i>key</i>     | storage key to be used for the I/O   |
| <i>flags</i>   | additional flags; defines the action to be performed for I/O processing.   |

## Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered).

## Returns

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in *lpm* is operational; -ENODEV, if the device is not operational.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_start_timeout_key` — start a s390 channel program with timeout and key

## Synopsis

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

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | target ccw device  |
| <i>cpa</i>     | logical start address of channel program   |
| <i>intparm</i> | user specific interruption parameter; will be presented back to <i>cdev</i> 's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request. |
| <i>lpm</i>     | defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.  |
| <i>key</i>     | storage key to be used for the I/O   |
| <i>flags</i>   | additional flags; defines the action to be performed for I/O processing.   |
| <i>expires</i> | timeout value in jiffies   |

## Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by *expires*. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing `ERR_PTR(-ETIMEDOUT)`.

## Returns

0, if the operation was successful; `-EBUSY`, if the device is busy, or status pending; `-EACCES`, if no path specified in *lpm* is operational; `-ENODEV`, if the device is not operational.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_start` — start a s390 channel program

## Synopsis

```
int ccw_device_start (struct ccw_device * cdev, struct ccw1 * cpa,  
unsigned long intparm, __u8 lpm, unsigned long flags);
```

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | target ccw device  |
| <i>cpa</i>     | logical start address of channel program   |
| <i>intparm</i> | user specific interruption parameter; will be presented back to <i>cdev</i> 's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request. |
| <i>lpm</i>     | defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.  |
| <i>flags</i>   | additional flags; defines the action to be performed for I/O processing.   |

## Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered).

## Returns

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in *lpm* is operational; -ENODEV, if the device is not operational.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_start_timeout` — start a s390 channel program with timeout

## Synopsis

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

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | target ccw device  |
| <i>cpa</i>     | logical start address of channel program   |
| <i>intparm</i> | user specific interruption parameter; will be presented back to <i>cdev</i> 's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request. |
| <i>lpm</i>     | defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.  |
| <i>flags</i>   | additional flags; defines the action to be performed for I/O processing.   |
| <i>expires</i> | timeout value in jiffies   |

## Description

Start a S/390 channel program. When the interrupt arrives, the IRQ handler is called, either immediately, delayed (dev-end missing, or sense required) or never (no IRQ handler registered). This function notifies the device driver if the channel program has not completed during the time specified by *expires*. If a timeout occurs, the channel program is terminated via xsch, hsch or csch, and the device's interrupt handler will be called with an irb containing ERR\_PTR(-ETIMEDOUT).

## Returns

0, if the operation was successful; -EBUSY, if the device is busy, or status pending; -EACCES, if no path specified in *lpm* is operational; -ENODEV, if the device is not operational.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_halt` — halt I/O request processing

## Synopsis

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

## Arguments

*cdev*        target ccw device

*intparm*    interruption parameter; value is only used if no I/O is outstanding, otherwise the `intparm` associated with the I/O request is returned

## Description

`ccw_device_halt` calls `hsch` on *cdev*'s subchannel.

## Returns

0 on success, `-ENODEV` on device not operational, `-EINVAL` on invalid device state, `-EBUSY` on device busy or interrupt pending.

## Context

Interrupts disabled, ccw device lock held



## Name

`ccw_device_resume` — resume channel program execution

## Synopsis

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

## Arguments

*cdev*   target ccw device

## Description

`ccw_device_resume` calls `rsch` on *cdev*'s subchannel.

## Returns

0 on success, `-ENODEV` on device not operational, `-EINVAL` on invalid device state, `-EBUSY` on device busy or interrupt pending.

## Context

Interrupts disabled, ccw device lock held

## Name

`ccw_device_get_ciw` — Search for CIW command in extended sense data.

## Synopsis

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

## Arguments

*cdev*    ccw device to inspect

*ct*      command type to look for

## Description

During SenseID, command information words (CIWs) describing special commands available to the device may have been stored in the extended sense data. This function searches for CIWs of a specified command type in the extended sense data.

## Returns

NULL if no extended sense data has been stored or if no CIW of the specified command type could be found, else a pointer to the CIW of the specified command type.

## Name

`ccw_device_get_path_mask` — get currently available paths

## Synopsis

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

## Arguments

*cdev* ccw device to be queried

## Returns

0 if no subchannel for the device is available, else the mask of currently available paths for the ccw device's subchannel.

## Name

`ccw_device_get_chp_desc` — return newly allocated channel-path descriptor

## Synopsis

```
struct channel_path_desc * ccw_device_get_chp_desc (struct ccw_device  
* cdev, int chp_idx);
```

## Arguments

*cdev*        device to obtain the descriptor for

*chp\_idx*    index of the channel path

## Description

On success return a newly allocated copy of the channel-path description data associated with the given channel path. Return `NULL` on error.

## Name

`ccw_device_get_id` — obtain a ccw device id

## Synopsis

```
void ccw_device_get_id (struct ccw_device * cdev, struct ccw_dev_id *  
dev_id);
```

## Arguments

*cdev*      device to obtain the id for

*dev\_id*    where to fill in the values

## Name

`ccw_device_tm_start_key` — perform start function

## Synopsis

```
int ccw_device_tm_start_key (struct ccw_device * cdev, struct tcw * tcw,
unsigned long intparm, u8 lpm, u8 key);
```

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | ccw device on which to perform the start function            |
| <i>tcw</i>     | transport-command word to be started                         |
| <i>intparm</i> | user defined parameter to be passed to the interrupt handler |
| <i>lpm</i>     | mask of paths to use   |
| <i>key</i>     | storage key to use for storage access                        |

## Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

## Name

`ccw_device_tm_start_timeout_key` — perform start function

## Synopsis

```
int ccw_device_tm_start_timeout_key (struct ccw_device * cdev, struct
tcw * tcw, unsigned long intparm, u8 lpm, u8 key, int expires);
```

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | ccw device on which to perform the start function            |
| <i>tcw</i>     | transport-command word to be started                         |
| <i>intparm</i> | user defined parameter to be passed to the interrupt handler |
| <i>lpm</i>     | mask of paths to use   |
| <i>key</i>     | storage key to use for storage access                        |
| <i>expires</i> | time span in jiffies after which to abort request            |

## Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

## Name

`ccw_device_tm_start` — perform start function

## Synopsis

```
int ccw_device_tm_start (struct ccw_device * cdev, struct tcw * tcw,
unsigned long intparm, u8 lpm);
```

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | ccw device on which to perform the start function            |
| <i>tcw</i>     | transport-command word to be started                         |
| <i>intparm</i> | user defined parameter to be passed to the interrupt handler |
| <i>lpm</i>     | mask of paths to use   |

## Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.



## Name

`ccw_device_tm_start_timeout` — perform start function

## Synopsis

```
int ccw_device_tm_start_timeout (struct ccw_device * cdev, struct tcw  
* tcw, unsigned long intparm, u8 lpm, int expires);
```

## Arguments

|                |  |
|----------------|--|
| <i>cdev</i>    | ccw device on which to perform the start function            |
| <i>tcw</i>     | transport-command word to be started                         |
| <i>intparm</i> | user defined parameter to be passed to the interrupt handler |
| <i>lpm</i>     | mask of paths to use   |
| <i>expires</i> | time span in jiffies after which to abort request            |

## Description

Start the tcw on the given ccw device. Return zero on success, non-zero otherwise.

## Name

`ccw_device_get_mdc` — accumulate max data count

## Synopsis

```
int ccw_device_get_mdc (struct ccw_device * cdev, u8 mask);
```

## Arguments

*cdev* ccw device for which the max data count is accumulated

*mask* mask of paths to use

## Description

Return the number of 64K-bytes blocks all paths at least support for a transport command. Return values  $\leq 0$  indicate failures.

## Name

`ccw_device_tm_intrg` — perform interrogate function

## Synopsis

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

## Arguments

*cdev* ccw device on which to perform the interrogate function

## Description

Perform an interrogate function on the given ccw device. Return zero on success, non-zero otherwise.

## Name

`ccw_device_get_schid` — obtain a subchannel id

## Synopsis

```
void ccw_device_get_schid (struct ccw_device * cdev, struct subchan-  
nel_id * schid);
```

## Arguments

*cdev*     device to obtain the id for

*schid*   where to fill in the values

# The channel-measurement facility

The channel-measurement facility provides a means to collect measurement data which is made available by the channel subsystem for each channel attached device.

## Name

arch/s390/include/asm/cmb.h — Document generation inconsistency

## Oops

### Warning

The template for this document tried to insert the structured comment from the file `arch/s390/include/asm/cmb.h` at this point, but none was found. This dummy section is inserted to allow generation to continue.

## Name

`enable_cmf` — switch on the channel measurement for a specific device

## Synopsis

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

## Arguments

*cdev* The ccw device to be enabled

## Description

Returns 0 for success or a negative error value.

## Note

If this is called on a device for which channel measurement is already enabled a reset of the measurement data is triggered.

## Context

non-atomic

## Name

`disable_cmf` — switch off the channel measurement for a specific device

## Synopsis

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

## Arguments

*cdev* The ccw device to be disabled

## Description

Returns 0 for success or a negative error value.

## Context

non-atomic

## Name

`cmf_read` — read one value from the current channel measurement block

## Synopsis

```
u64 cmf_read (struct ccw_device * cdev, int index);
```

## Arguments

*cdev*     the channel to be read

*index*   the index of the value to be read

## Description

Returns the value read or 0 if the value cannot be read.

## Context

any



## Name

`cmf_readall` — read the current channel measurement block

## Synopsis

```
int cmf_readall (struct ccw_device * cdev, struct cmbdata * data);
```

## Arguments

*cdev* the channel to be read

*data* a pointer to a data block that will be filled

## Description

Returns 0 on success, a negative error value otherwise.

## Context

any

---

## Chapter 3. The ccwgroup bus

The ccwgroup bus only contains artificial devices, created by the user. Many networking devices (e.g. qeth) are in fact composed of several ccw devices (like read, write and data channel for qeth). The ccwgroup bus provides a mechanism to create a meta-device which contains those ccw devices as slave devices and can be associated with the netdevice.

### ccw group devices

## Name

struct ccwgroup\_device — ccw group device

## Synopsis

```
struct ccwgroup_device {
    enum state;
    unsigned int count;
    struct device dev;
    struct work_struct ungroup_work;
    struct ccw_device * cdev[0];
};
```

## Members

|              |   |
|--------------|---|
| state        | online/offline state  |
| count        | number of attached slave devices  |
| dev          | embedded device structure   |
| ungroup_work | work to be done when a ccwgroup notifier has action type BUS_NOTIFY_UNBIND_DRIVER |
| cdev[0]      | variable number of slave devices, allocated as needed                             |

## Name

struct ccwgroup\_driver — driver for ccw group devices

## Synopsis

```
struct ccwgroup_driver {
    int (* setup) (struct ccwgroup_device *);
    void (* remove) (struct ccwgroup_device *);
    int (* set_online) (struct ccwgroup_device *);
    int (* set_offline) (struct ccwgroup_device *);
    void (* shutdown) (struct ccwgroup_device *);
    int (* prepare) (struct ccwgroup_device *);
    void (* complete) (struct ccwgroup_device *);
    int (* freeze) (struct ccwgroup_device *);
    int (* thaw) (struct ccwgroup_device *);
    int (* restore) (struct ccwgroup_device *);
    struct device_driver driver;
};
```

## Members

|             |  |
|-------------|--|
| setup       | function called during device creation to setup the device |
| remove      | function called on remove                                  |
| set_online  | function called when device is set online                  |
| set_offline | function called when device is set offline                 |
| shutdown    | function called when device is shut down                   |
| prepare     | prepare for pm state transition                            |
| complete    | undo work done in <i>prepare</i>                           |
| freeze      | callback for freezing during hibernation snapshotting      |
| thaw        | undo work done in <i>freeze</i>                            |
| restore     | callback for restoring after hibernation                   |
| driver      | embedded driver structure                                  |

## Name

ccwgroup\_set\_online — enable a ccwgroup device

## Synopsis

```
int ccwgroup_set_online (struct ccwgroup_device * gdev);
```

## Arguments

*gdev* target ccwgroup device

## Description

This function attempts to put the ccwgroup device into the online state.

## Returns

0 on success and a negative error value on failure.

## Name

`ccwgroup_set_offline` — disable a ccwgroup device

## Synopsis

```
int ccwgroup_set_offline (struct ccwgroup_device * gdev);
```

## Arguments

*gdev* target ccwgroup device

## Description

This function attempts to put the ccwgroup device into the offline state.

## Returns

0 on success and a negative error value on failure.

## Name

`ccwgroup_create_dev` — create and register a ccw group device

## Synopsis

```
int ccwgroup_create_dev (struct device * parent, struct ccwgroup_driver  
* gdrv, int num_devices, const char * buf);
```

## Arguments

|                    |  |
|--------------------|--|
| <i>parent</i>      | parent device for the new device                           |
| <i>gdrv</i>        | driver for the new group device                            |
| <i>num_devices</i> | number of slave devices                                    |
| <i>buf</i>         | buffer containing comma separated bus ids of slave devices |

## Description

Create and register a new ccw group device as a child of *parent*. Slave devices are obtained from the list of bus ids given in *buf*.

## Returns

0 on success and an error code on failure.

## Context

non-atomic

## Name

ccwgroup\_driver\_register — register a ccw group driver

## Synopsis

```
int ccwgroup_driver_register (struct ccwgroup_driver * cdriver);
```

## Arguments

*cdriver* driver to be registered

## Description

This function is mainly a wrapper around `driver_register`.



## Name

`ccwgroup_driver_unregister` — deregister a ccw group driver

## Synopsis

```
void ccwgroup_driver_unregister (struct ccwgroup_driver * cdriver);
```

## Arguments

*cdriver* driver to be deregistered

## Description

This function is mainly a wrapper around `driver_unregister`.

## Name

`ccwgroup_probe_ccwdev` — probe function for slave devices

## Synopsis

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

## Arguments

*cdev* ccw device to be probed

## Description

This is a dummy probe function for ccw devices that are slave devices in a ccw group device.

## Returns

always 0

## Name

`ccwgroup_remove_ccwdev` — remove function for slave devices

## Synopsis

```
void ccwgroup_remove_ccwdev (struct ccw_device * cdev);
```

## Arguments

*cdev* ccw device to be removed

## Description

This is a remove function for ccw devices that are slave devices in a ccw group device. It sets the ccw device offline and also deregisters the embedding ccw group device.

---

## Chapter 4. Generic interfaces

Some interfaces are available to other drivers that do not necessarily have anything to do with the busses described above, but still are indirectly using basic infrastructure in the common I/O layer. One example is the support for adapter interrupts.

## Name

`register_adapter_interrupt` — register adapter interrupt handler

## Synopsis

```
int register_adapter_interrupt (struct airq_struct * airq);
```

## Arguments

*airq* pointer to adapter interrupt descriptor

## Description

Returns 0 on success, or -EINVAL.

## Name

`unregister_adapter_interrupt` — unregister adapter interrupt handler

## Synopsis

```
void unregister_adapter_interrupt (struct airq_struct * airq);
```

## Arguments

*airq* pointer to adapter interrupt descriptor

## Name

`airq_iv_create` — create an interrupt vector

## Synopsis

```
struct airq_iv * airq_iv_create (unsigned long bits, unsigned long  
flags);
```

## Arguments

*bits*    number of bits in the interrupt vector

*flags*   allocation flags

## Description

Returns a pointer to an interrupt vector structure

## Name

`airq_iv_release` — release an interrupt vector

## Synopsis

```
void airq_iv_release (struct airq_iv * iv);
```

## Arguments

*iv* pointer to interrupt vector structure



## Name

`airq_iv_alloc` — allocate irq bits from an interrupt vector

## Synopsis

```
unsigned long airq_iv_alloc (struct airq_iv * iv, unsigned long num);
```

## Arguments

*iv* pointer to an interrupt vector structure

*num* number of consecutive irq bits to allocate

## Description

Returns the bit number of the first irq in the allocated block of irqs, or -1UL if no bit is available or the `AIRQ_IV_ALLOC` flag has not been specified

## Name

`airq_iv_free` — free irq bits of an interrupt vector

## Synopsis

```
void airq_iv_free (struct airq_iv * iv, unsigned long bit, unsigned  
long num);
```

## Arguments

*iv* pointer to interrupt vector structure

*bit* number of the first irq bit to free

*num* number of consecutive irq bits to free

## Name

`airq_iv_scan` — scan interrupt vector for non-zero bits

## Synopsis

```
unsigned long airq_iv_scan (struct airq_iv * iv, unsigned long start,  
unsigned long end);
```

## Arguments

*iv*       pointer to interrupt vector structure

*start*   bit number to start the search

*end*     bit number to end the search

## Description

Returns the bit number of the next non-zero interrupt bit, or -1UL if the scan completed without finding any more any non-zero bits.