Generic Counter Interface

Introduction

Counter devices are prevalent among a diverse spectrum of industries. The ubiquitous presence of these devices necessitates a common interface and standard of interaction and exposure. This driver API attempts to resolve the issue of duplicate code found among existing counter device drivers by introducing a generic counter interface for consumption. The Generic Counter interface enables drivers to support and expose a common set of components and functionality present in counter devices.

Theory

Counter devices can vary greatly in design, but regardless of whether some devices are quadrature encoder counters or tally counters, all counter devices consist of a core set of components. This core set of components, shared by all counter devices, is what forms the essence of the Generic Counter interface.

There are three core components to a counter:

- Signal: Stream of data to be evaluated by the counter.
- Synapse: Association of a Signal, and evaluation trigger, with a Count.
- Count: Accumulation of the effects of connected Synapses.

SIGNAL

A Signal represents a stream of data. This is the input data that is evaluated by the counter to determine the count data; e.g. a quadrature signal output line of a rotary encoder. Not all counter devices provide user access to the Signal data, so exposure is optional for drivers.

When the Signal data is available for user access, the Generic Counter interface provides the following available signal values:

- SIGNAL_LOW: Signal line is in a low state.
- SIGNAL_HIGH: Signal line is in a high state.

A Signal may be associated with one or more Counts.

SYNAPSE

A Synapse represents the association of a Signal with a Count. Signal data affects respective Count data, and the Synapse represents this relationship.

The Synapse action mode specifies the Signal data condition that triggers the respective Count's count function evaluation to update the count data. The Generic Counter interface provides the following available action modes:

- None: Signal does not trigger the count function. In Pulse-Direction count function mode, this Signal is evaluated as Direction.
- Rising Edge: Low state transitions to high state.
- Falling Edge: High state transitions to low state.
- Both Edges: Any state transition.

A counter is defined as a set of input signals associated with count data that are generated by the evaluation of the state of the associated input signals as defined by the respective count functions. Within the context of the Generic Counter interface, a counter consists of Counts each associated with a set of Signals, whose respective Synapse instances represent the count function update conditions for the associated Counts.

A Synapse associates one Signal with one Count.

COUNT

A Count represents the accumulation of the effects of connected Synapses; i.e. the count data for a set of Signals. The Generic Counter interface represents the count data as a natural number.

A Count has a count function mode which represents the update behavior for the count data. The Generic Counter interface provides the following available count function modes:

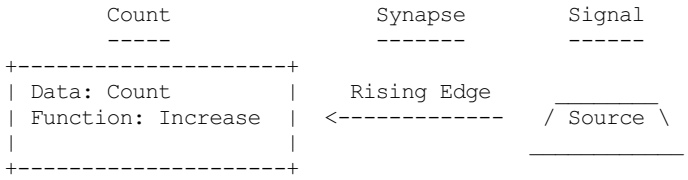
- Increase: Accumulated count is incremented.
- Decrease: Accumulated count is decremented.
- Pulse-Direction: Rising edges on signal A updates the respective count. The input level of signal B determines direction.
- Quadrature: A pair of quadrature encoding signals are evaluated to determine position and direction. The following Quadrature modes are available:
    - x1 A: If direction is forward, rising edges on quadrature pair signal A updates the respective count; if the direction is backward, falling edges on quadrature pair signal A updates the respective count. Quadrature encoding determines the direction.
    - x1 B: If direction is forward, rising edges on quadrature pair signal B updates the respective count; if the direction is

backward, falling edges on quadrature pair signal B updates the respective count. Quadrature encoding determines the direction.
    - x2 A: Any state transition on quadrature pair signal A updates the respective count. Quadrature encoding determines the direction.
    - x2 B: Any state transition on quadrature pair signal B updates the respective count. Quadrature encoding determines the direction.
    - x4: Any state transition on either quadrature pair signals updates the respective count. Quadrature encoding determines the direction.

A Count has a set of one or more associated Synapses.

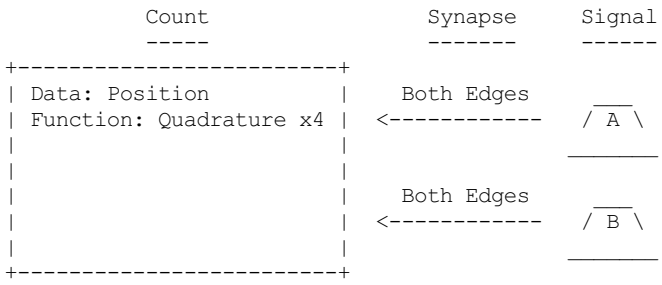
Paradigm

The most basic counter device may be expressed as a single Count associated with a single Signal via a single Synapse. Take for example a counter device which simply accumulates a count of rising edges on a source input line:



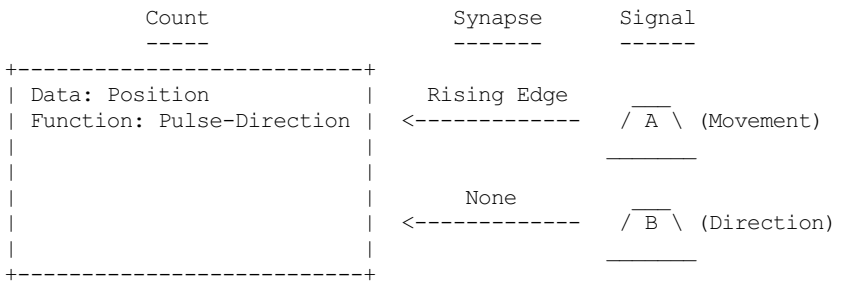
In this example, the Signal is a source input line with a pulsing voltage, while the Count is a persistent count value which is repeatedly incremented. The Signal is associated with the respective Count via a Synapse. The increase function is triggered by the Signal data condition specified by the Synapse -- in this case a rising edge condition on the voltage input line. In summary, the counter device existence and behavior is aptly represented by respective Count, Signal, and Synapse components: a rising edge condition triggers an increase function on an accumulating count datum.

A counter device is not limited to a single Signal; in fact, in theory many Signals may be associated with even a single Count. For example, a quadrature encoder counter device can keep track of position based on the states of two input lines:



In this example, two Signals (quadrature encoder lines A and B) are associated with a single Count: a rising or falling edge on either A or B triggers the "Quadrature x4" function which determines the direction of movement and updates the respective position data. The "Quadrature x4" function is likely implemented in the hardware of the quadrature encoder counter device; the Count, Signals, and Synapses simply represent this hardware behavior and functionality.

Signals associated with the same Count can have differing Synapse action mode conditions. For example, a quadrature encoder counter device operating in a non-quadrature Pulse-Direction mode could have one input line dedicated for movement and a second input line dedicated for direction:



Only Signal A triggers the "Pulse-Direction" update function, but the instantaneous state of Signal B is still required in order to know the direction so that the position data may be properly updated. Ultimately, both Signals are associated with the same Count via two respective Synapses, but only one Synapse has an active action mode condition which triggers the respective count function while the other is left with a "None" condition action mode to indicate its respective Signal's availability for state evaluation despite its non-triggering mode.

Keep in mind that the Signal, Synapse, and Count are abstract representations which do not need to be closely married to their respective physical sources. This allows the user of a counter to divorce themselves from the nuances of physical components (such as whether an input line is differential or single-ended) and instead focus on the core idea of what the data and process represent (e.g.

position as interpreted from quadrature encoding data).

Driver API

Driver authors may utilize the Generic Counter interface in their code by including the include/linux/counter.h header file. This header file provides several core data structures, function prototypes, and macros for defining a counter device.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst, line 234)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/counter.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst, line 237)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/counter/counter-core.c
   :export:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst, line 240)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/counter/counter-chrdev.c
   :export:
```

Driver Implementation

To support a counter device, a driver must first allocate the available Counter Signals via counter_signal structures. These Signals should be stored as an array and set to the signals array member of an allocated counter_device structure before the Counter is registered to the system.

Counter Counts may be allocated via counter_count structures, and respective Counter Signal associations (Synapses) made via counter_synapse structures. Associated counter_synapse structures are stored as an array and set to the synapses array member of the respective counter_count structure. These counter_count structures are set to the counts array member of an allocated counter_device structure before the Counter is registered to the system.

Driver callbacks must be provided to the counter_device structure in order to communicate with the device: to read and write various Signals and Counts, and to set and get the "action mode" and "function mode" for various Synapses and Counts respectively.

A counter_device structure is allocated using counter_alloc() and then registered to the system by passing it to the counter_add() function, and unregistered by passing it to the counter_unregister function. There are device managed variants of these functions: devm_counter_alloc() and devm_counter_add().

The struct counter_comp structure is used to define counter extensions for Signals, Synapses, and Counts.

The "type" member specifies the type of high-level data (e.g. BOOL, COUNT_DIRECTION, etc.) handled by this extension. The "`*_read`" and "`*_write`" members can then be set by the counter device driver with callbacks to handle that data using native C data types (i.e. u8, u64, etc.).

Convenience macros such as `COUNTER_COMP_COUNT_U64` are provided for use by driver authors. In particular, driver authors are expected to use the provided macros for standard Counter subsystem attributes in order to maintain a consistent interface for userspace. For example, a counter device driver may define several standard attributes like so:

```
struct counter_comp count_ext[] = {
        COUNTER_COMP_DIRECTION(count_direction_read),
        COUNTER_COMP_ENABLE(count_enable_read, count_enable_write),
        COUNTER_COMP_CEILING(count_ceiling_read, count_ceiling_write),
};
```

This makes it simple to see, add, and modify the attributes that are supported by this driver ("direction", "enable", and "ceiling") and to maintain this code without getting lost in a web of struct braces.

Callbacks must match the function type expected for the respective component or extension. These function types are defined in the struct counter_comp structure as the "`*_read`" and "`*_write`" union members.

The corresponding callback prototypes for the extensions mentioned in the previous example above would be:

```
int count_direction_read(struct counter_device *counter,
                         struct counter_count *count,
                         enum counter_count_direction *direction);
int count_enable_read(struct counter_device *counter,
                      struct counter_count *count, u8 *enable);
int count_enable_write(struct counter_device *counter,
                       struct counter_count *count, u8 enable);
int count_ceiling_read(struct counter_device *counter,
                       struct counter_count *count, u64 *ceiling);
int count_ceiling_write(struct counter_device *counter,
                        struct counter_count *count, u64 ceiling);
```

Determining the type of extension to create is a matter of scope.

- Signal extensions are attributes that expose information/control specific to a Signal. These types of attributes will exist under a Signal's directory in sysfs.

  For example, if you have an invert feature for a Signal, you can have a Signal extension called "invert" that toggles that feature: /sys/bus/counter/devices/counterX/signalY/invert

- Count extensions are attributes that expose information/control specific to a Count. These type of attributes will exist under a Count's directory in sysfs.

  For example, if you want to pause/unpause a Count from updating, you can have a Count extension called "enable" that toggles such: /sys/bus/counter/devices/counterX/countY/enable

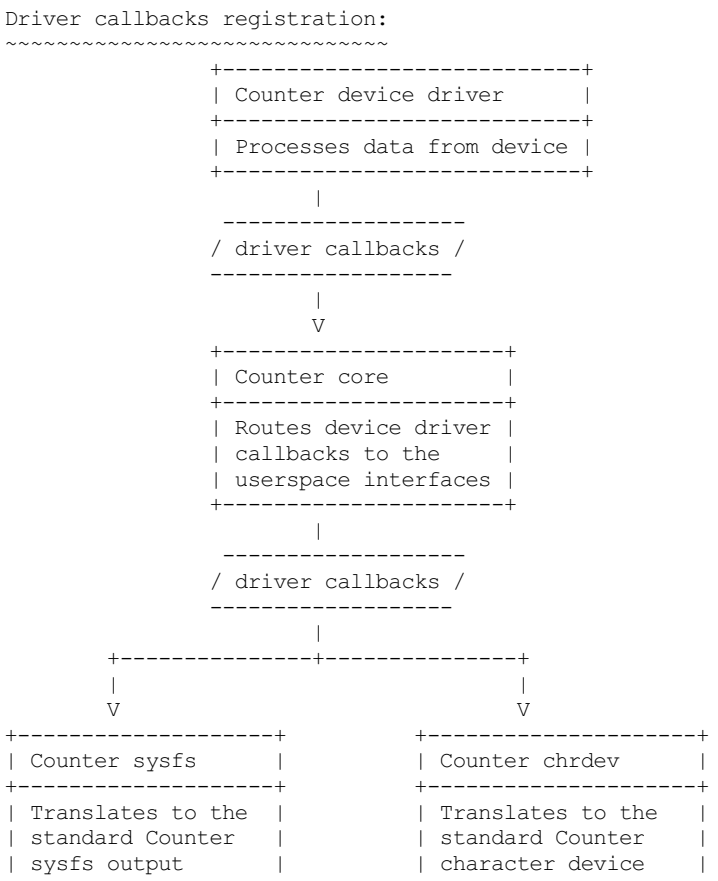
- Device extensions are attributes that expose information/control non-specific to a particular Count or Signal. This is where you would put your global features or other miscellaneous functionality.

  For example, if your device has an overtemp sensor, you can report the chip overheated via a device extension called "error_overtemp": /sys/bus/counter/devices/counterX/error_overtemp

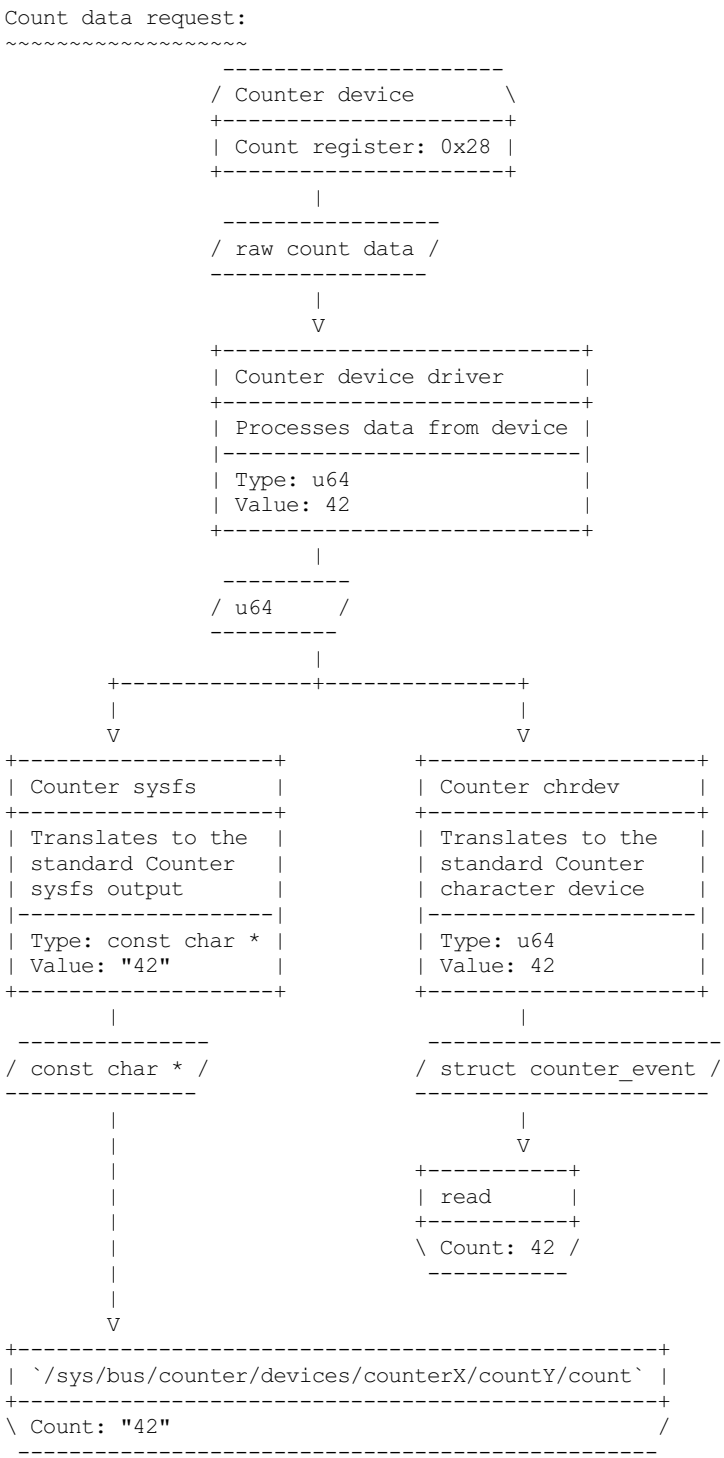
Subsystem Architecture

Counter drivers pass and take data natively (i.e. `u8`, `u64`, etc.) and the shared counter module handles the translation between the sysfs interface. This guarantees a standard userspace interface for all counter drivers, and enables a Generic Counter chrdev interface via a generalized device driver ABI.

A high-level view of how a count value is passed down from a counter driver is exemplified by the following. The driver callbacks are first registered to the Counter core component for use by the Counter userspace interface components:



Thereafter, data can be transferred directly between the Counter device driver and Counter userspace interface:



There are four primary components involved:

Counter device driver

Communicates with the hardware device to read/write data; e.g. counter drivers for quadrature encoders, timers, etc.

Counter core

Registers the counter device driver to the system so that the respective callbacks are called during userspace interaction.

Counter sysfs

Translates counter data to the standard Counter sysfs interface format and vice versa.

Please refer to the `Documentation/ABI/testing/sysfs-bus-counter` file for a detailed breakdown of the available Generic Counter interface sysfs attributes.

Counter chrdev

Translates Counter events to the standard Counter character device; data is transferred via standard character device read calls, while Counter events are configured via ioctl calls.

Sysfs Interface

Several sysfs attributes are generated by the Generic Counter interface, and reside under the `/sys/bus/counter/devices/counterX` directory, where `X` is to the respective counter device id. Please see `Documentation/ABI/testing/sysfs-bus-counter` for detailed information on each Generic Counter interface sysfs attribute.

Through these sysfs attributes, programs and scripts may interact with the Generic Counter paradigm Counts, Signals, and Synapses of respective counter devices.

Counter Character Device

Counter character device nodes are created under the `/dev` directory as `counterX`, where `X` is the respective counter device id. Defines for the standard Counter data types are exposed via the userspace `include/uapi/linux/counter.h` file.

Counter events

Counter device drivers can support Counter events by utilizing the `counter_push_event` function:

```
void counter_push_event(struct counter_device *const counter, const u8 event,
                        const u8 channel);
```

The event id is specified by the `event` parameter; the event channel id is specified by the `channel` parameter. When this function is called, the Counter data associated with the respective event is gathered, and a `struct counter_event` is generated for each datum and pushed to userspace.

Counter events can be configured by users to report various Counter data of interest. This can be conceptualized as a list of Counter component read calls to perform. For example:

<table>
<tr><th>COUNTER_EVENT_OVERFLOW</th><th>COUNTER_EVENT_INDEX</th></tr>
<tr><td>Channel 0</td><td>Channel 0</td></tr>
<tr><td rowspan="2">Count 0
Count 1
Signal 3
Count 4 Extension 2
Signal 5 Extension 0</td><td>Signal 0
Signal 0 Extension 0
Extension 4</td></tr>
<tr><td>Channel 1</td></tr>
<tr><td></td><td>Signal 4
Signal 4 Extension 0
Count 7</td></tr>
</table>

When `counter_push_event(counter, COUNTER_EVENT_INDEX, 1)` is called for example, it will go down the list for the `COUNTER_EVENT_INDEX` event channel 1 and execute the read callbacks for Signal 4, Signal 4 Extension 0, and Count 7 -- the data returned for each is pushed to a kfifo as a `struct counter_event`, which userspace can retrieve via a standard read operation on the respective character device node.

Userspace

Userspace applications can configure Counter events via ioctl operations on the Counter character device node. There following ioctl codes are supported and provided by the `linux/counter.h` userspace header file:

- :c:macro:`COUNTER_ADD_WATCH_IOCTL`

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst`, **line 548**); *backlink*

  Unknown interpreted text role "c:macro".

- :c:macro:`COUNTER_ENABLE_EVENTS_IOCTL`

  System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst`, **line 550**); *backlink*

  Unknown interpreted text role "c:macro".

- :c:macro:`COUNTER_DISABLE_EVENTS_IOCTL`

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master)(Documentation)(driver-api)generic-counter.rst`, line 552); *backlink*
>
> Unknown interpreted text role "c:macro".

To configure events to gather Counter data, users first populate a `struct counter_watch` with the relevant event id, event channel id, and the information for the desired Counter component from which to read, and then pass it via the `COUNTER_ADD_WATCH_IOCTL` ioctl command.

Note that an event can be watched without gathering Counter data by setting the `component.type` member equal to `COUNTER_COMPONENT_NONE`. With this configuration the Counter character device will simply populate the event timestamps for those respective `struct counter_event` elements and ignore the component value.

The `COUNTER_ADD_WATCH_IOCTL` command will buffer these Counter watches. When ready, the `COUNTER_ENABLE_EVENTS_IOCTL` ioctl command may be used to activate these Counter watches.

Userspace applications can then execute a `read` operation (optionally calling `poll` first) on the Counter character device node to retrieve `struct counter_event` elements with the desired data.