

# Linux Utils Internals

---

## Contents

---

### LinuxUtils Services

- CMEM
- EDMAK
  - EDMAK Resource Types
  - Supported EDMAK Services
  - EDMA\_IOCREQUESTDMA
  - EDMA\_IOCREGUSER
  - EDMA\_IOCRELEASEDMA
  - EDMA\_GETBASEPHYSADDR
  - EDMA\_GETVERSION
- IRQK/VICP
  - Supported VICP Resources
  - IRQK interrupt handler
  - Resource-to-interrupt table
  - FLEXIBLE vs FIXED
  - Supported IRQK Services
  - IRQK\_IOCREQUESTIRQ
  - IRQK\_IOCRELEASEIRQ
  - IRQK\_IOCWAIT
  - IRQK\_IOCDONE
  - Miscellaneous Stuff
- VICP User Interface
  - Supported VICP Resources
- Future LinuxUtils Development
- CMEM

## LinuxUtils Services

LinuxUtils is a set of utility middleware that provides access to certain Linux kernel services for user programs. All the services share a common goal - providing access to kernel resources in coordination with other independent system requestors. LinuxUtils provides access by way of the Linux kernel services and generic Unix interfaces, and the services provided by LinuxUtils can be ported to other OS foundations.

The lowest layer of these services is the kernel module (device driver). Each kernel module has a user-level component that provides easier-to-understand APIs to encapsulate the more complicated ioctl() interface to the kernel module services and allow the underlying interface to change without changing the user-level API. Most LinuxUtils kernel modules have equivalently-named user-level layers, except for the irqk kernel module which is called by the vicp user layer. In the descriptions that follow, the module name will be suffixed with 'k' in order to denote the kernel portion of the module. Most user-level APIs are simply shims on top of the corresponding ioctl() to the kernel, but some include functionality in addition to the ioctl() call.

Ultimately it is the user-level APIs that need to be supported since they are the interface to the user application. The boundary between user-level and kernel processing can move either way depending on your underlying OS capability, and it is generally preferred to put as much in the user layer as possible.

Whenever possible, all Linux Utils services will attempt to "clean up" resources when it is detected that a requestor of a resource has terminated without explicitly releasing the resource. This is accomplished by attaching the user's file handle for the module's device driver to the resource, so that when that handle is closed all the resources belonging to that handle can be automatically freed. In addition to that, all resources that were granted by the kernel are released when the module's device driver is removed from the system.

As LinuxUtils has developed over the years it has had to keep up with changes to the Linux kernel API set, while maintaining support for the older kernels on which it was developed. Therefore, most of the LinuxUtils kernel module C files have #ifdef's based on the kernel version against which the module is built. One particular #ifdef case is not based on the Linux kernel version, per se. The macro LSP\_210 is manually defined (either on the command line or in a C file itself) when using the LSP 2.10 release. This is because there is parallel development of Linux with common number versioning. For instance, the EDMA interface for LSP\_210 doesn't exist in non-LSP releases of the Linux kernel that use the same kernel versioning.

## CMEM

---

CMEM, a manager for Contiguous Memory, was the initial component in Linux Utils and is the most widely used. [There are details about CMEM, on the CMEM Overview article.](#)

## EDMAK

---

The edmak kernel module provides resource management for the EDMA HW to ARM-based Linux user applications. It is used only for resource management (allocation & freeing) and does not provide any functional layer for programming the EDMA operation. The functional operation of the EDMA resource is intended to be carried out by a user-level component, which is enabled by virtue of a memory-mapped pointer to the allocated resource provided by the EDMA user layer (and supported by a kernel module service).

The interface is geared toward the particular architecture of the EDMA. The interface contains elements for requesting particular types of EDMA resources, such as TCCs, PaRAM entries, and EDMA channels themselves. Please see the EDMA architecture for a description of EDMA resources and terminology.

This interface supports multiple simultaneous users of the same resource by way of a "register" API.

EDMAK Resource Types

The following descriptions use the term "channel" to indicate an EDMA channel with an associated TCC and PaRAM which are all used in conjunction for driving a DMA transfer. For the EDMA architecture there are also PaRAM entries that don't have a corresponding EDMA channel or TCC. These unattached PaRAM entries are used when setting up a linked transfer for a particular channel, and can be requested independently of any channel, including support for requesting multiple consecutive unattached PaRAM entries. The current DM365 Linux kernel refers to these types of PaRAMs as "slots".

EDMA can be used to request the following types of EDMA resources:

- TCCs (any OR symmetrical to the related channel)
  - PaRAMs ([multiple consecutive of] any OR fixed OR fixed-but-not-exact)
  - channels (any OR fixed)
  - QDMAs (any OR fixed)

These resource requests are translated into the particular form for the underlying kernel request support. For example, for DM365 the Linux kernel APIs that are used for requesting resources are `edma_alloc_channel()` and `edma_alloc_cont_slots()`. As of this writing, `edma_alloc_channel()` is somewhat limited in the range of resources that can be requested, and as a result some of the types of requested resources cannot be honored. When a valid request can't be honored, `edmak.ko` will return a code indicating that the request is invalid.

The following table shows which function parameters are valid for all types of allocations:

devId	input parameters	output parameters
EDMA#	none	channel, tcc, param*#
EDMA_EDMAANY	tcc	channel, tcc, param*
EDMA_QDMA#	tcc	channel, tcc, param
EDMA_QDMAANY	tcc	channel, tcc, param
EDMA_PARAMANY	nParams	channel, tcc, param*

```
* channel and param are the same value
# channel and tcc are the same value
```

Supported EDMAK Services

ioctl() cmd	related user-level API
EDMA_IOCREQUESTDMA	EDMA_getResource()
EDMA_IOCREGUSER	EDMA_registerResource()
EDMA_IOCRELEASEDMA	EDMA_freeResource()/EDMA_unregister()
EDMA_IOCGETBASEPHYSADDR	EDMA_mapBaseAddress()
EDMA_IOCGETVERSION	EDMA_getVersion()/EDMA_init()

EDMA\_IOCREQUESTDMA

This command lies at the heart of the EDMAK service. All resources are requested through it. The parameters for the request are passed through the structure `EDMA_requestDmaParams` and are interpreted according to the table above.

For all resource types except `EDMA_PARAMANY` the kernel API `edma_alloc_channel()` is used to request the resource. For `EDMA_PARAMANY`, `edma_alloc_cont_slots()` is used.

`edma_alloc_channel()` does not support QDMA allocation, so a request for a QDMA resource is rejected by `edmak`. It also does not support requesting a channel with `TCCANY`, since `TCCSYMM` is implied (in other words, contrary to the table above, 'tcc' is not an input to `EDMAANY`, since you will be granted the TCC that corresponds to the granted channel).

In addition to requesting the resource, this command will initialize the registration list with the requestor as the lone registered user. More requestors can be registered with ...

EDMA\_IOCREGUSER

This command will simply add the requestor to the already-allocated resource user list. For Unix-based systems, the "requestor" is the user-level file handle for the module's device driver, which has the benefit of allowing different threads of a process to manage it since all threads in a process will share a common file handle.

If the requested resource is not already allocated, an error shall result.

EDMA\_IOCRELEASEDMA

This command will release a resource "registration", and if there are no more registered users then the resource itself will be freed back to the kernel.

It is used for both the `EDMA_freeResource()` and the `EDMA_unregister()` user APIs (more precisely, `EDMA_unregister()` directly calls `EDMA_freeResource()`).

EDMA\_GETBASEPHYSADDR

This command retrieves the base physical address for the set of EDMA HW registers. This address can then be used by the user API `EDMA_mapBaseAddress()` to map this address range into the calling process' address space, so that the calling process can subsequently directly access any EDMA register (however, the calling process should play fair and access only the registers related to their granted resources).

EDMA\_GETVERSION

This command retrieves the user-kernel interface version from the `edmak` module, for the purpose of comparing it to the user-kernel interface version with which the user layer was built. It is important that they match to ensure that the kernel module interprets the commands properly.

IRQK/VICP

The `irqk` kernel module provides resource management for IRQs (interrupts) that are needed by VICP/HDVICP/MJCP coprocessors. It is the underlying kernel module for the user layer VICP (which supports VICP/HDVICP/MJCP). It accepts requests based on high-level coprocessor resource names and translates those resource names to the corresponding IRQ for the system in use. There is support for different "types" of IRQs for a given resource.

It also has a minimal "functional" role for IRQs - there is a command that allows a user process to wait on the completion of the requested IRQ.

This interface supports multiple simultaneous users of the same resource by way of a "register" API.

Supported VICP Resources

- VICP\_IMX0
- VICP\_IMX1
- VICP\_MJCP
- VICP\_NSF
- VICP\_HDVICP0
- VICP\_HDVICP1

The user layer VICP is used to specify which resource to manage. IRQK translates each VICP resource into the corresponding interrupt number on the host, and uses that interrupt number for subsequent use in kernel APIs.

Each of the resources listed above has one or more interrupts with which it is associated. For resources that have more than one supported interrupt, the user application is responsible for configuring the coprocessor to generate the particular one requested through VICP.

The source of the interrupt is chosen through VICP attributes, and can be either `VICP_FIXED` or `VICP_FLEXIBLE`. `VICP_FIXED` refers to a dedicated assignment of the resources' interrupt source to a particular IRQ, in a one-to- one fashion. `VICP_FLEXIBLE` refers to a flexible assignment, where any one of a set of resources can generate the particular IRQ. If a particular source uses an IRQ that is multiplexed, then IRQK will program the appropriate MUX register for the requested source.

It is important to note the distinction between `VICP_FLEXIBLE` and an IRQ that can have multiple sources. The term "FLEXIBLE" is referring to the fact that a dedicated source of an IRQ can have multiple generators (or, in other terminology, triggers), whereas an IRQ with multiple sources responds to one of the sources depending on the interrupt multiplexor setting. For `FLEXIBLE`, an unrelated coprocessor setting controls the source, whereas for multiplexed IRQs, IRQK controls the source by way of the IRQ-related interrupt multiplexor.

IRQK interrupt handler

The IRQK module uses a general interrupt handler for all IRQs that it services. This handler simply posts a semaphore related to the IRQ, and a user application can wait on this semaphore through one of IRQK's `ioctl()` commands.

Resource-to-interrupt table

LinuxUtils' IRQK module defines a two-dimensional table for performing the translation from resource to interrupt. This table is setup during module initialization. The table's first dimension is indexed with the resource type, and its 2nd dimension is indexed with the resource's interrupt source - either `IRQK_FLEXIBLELINE` or `IRQ_FIXEDLINE`.

Also in this table are possible interrupt multiplexor values, if a particular interrupt can be switched to be driven by more than one resource. For instance, on a DM365 the NSF interrupt number (`IRQ_DM365_INSFINT`) can be driven by either the NSF resource or `VPSS_INT7` (not a resource managed by IRQK), and when requested through IRQK, IRQK will write the appropriate value into the `IRQ_DM365_INSFINT` field of the interrupt multiplexor register to cause the NSF resource to drive the IRQ. Note that is different than the previously mentioned interrupt source, whereby the coprocessor is programmed to generate an interrupt based on one of many possible sources, all going to a particular IRQ.

FLEXIBLE vs FIXED

The term `FLEXIBLE` here refers to the resource's non-unique interrupt source. For instance, for the DM365 the resources `NSF/IMX0/IMX1` can all generate the interrupt `IRQ_DM365_IMCOPINT` as their `FLEXIBLE` source, whereas for their fixed source these resources generate the following interrupts:

resource	FIXED interrupt
NSF	IRQ_DM365_INSFINT
IMX0	IRQ_DM365_IMXINT0
IMX1	IRQ_DM365_IMXINT1

Note that the interrupt symbols used above are as defined by the Linux kernel source code, and they are defined simply as an interrupt number.

Supported IRQK Services

ioctl() cmd	related user-level API

IRQK_IOCREQUESTIRQ	VICP_register()
IRQK_IOCRELEASEIRQ	VICP_unregister()
IRQK_IOCWAIT	VICP_wait()
IRQK_IOCDONE	VICP_done()

IRQK\_IOCREQUESTIRQ

This command handles requests for particular IRQs related to the VICP resource that is requested. It maps this request through a table that is initialized during module initialization, based on the HW device. It calls the Linux kernel's request\_irq() API to acquire the interrupt and register IRQK's handler. It then programs the interrupt multiplexor register if the resource has a setting for that. Finally, it adds the requestor to the list of registered owners.

If the resource is already held by a different requestor, the new requestor is placed on the list of resource users, becoming equivalent in ownership to the original requestor.

It's important to note that it is used for requesting IRQs only for the supported resources, since it bases its request on the resource name and not some general IRQ number.

If the requested resource's interrupt is shared by a different resource (and therefore has a bit for the interrupt multiplexor) and that different resource currently owns the IRQ, then this command will wait for the holder of the IRQ to release it. Upon getting the resource after it is released, the interrupt multiplexor will be programmed for use with the new resource.

IRQK\_IOCRELEASEIRQ

This command removes the caller from the list of registered owners of the resource, and if no more registered owners are on the list it releases the resource altogether. This involves releasing the IRQ back to the Linux kernel and re-initializing the resource's data structure.

If the requested resource's interrupt is shared by a different resource (and therefore has a bit for the interrupt multiplexor) then the interrupt multiplexor is programmed with the original, pre-request state for the resource's IRQ.

IRQK\_IOCWAIT

This command simply pends on a resource's IRQ semaphore. The IRQK IRQ handler posts a semaphore related to the resource. This means that if the interrupt happens before the wait then the wait will simply fall through, but if the wait happens before the interrupt, the caller will block until the interrupt posts the semaphore to unblock the caller. Since a counting semaphore is used, each post will accumulate in the semaphore if no wait is requested between interrupts.

IRQK\_IOCDONE

This command simply returns the state of the IRQ since the time it was requested or last waited on. It is basically a flag indicating if the resource's IRQ has fired. It is cleared upon the completion of the IRQK\_IOCWAIT command, so as to allow fresh reporting of a new IRQ.

Miscellaneous Stuff

One of the more ambiguous aspects of the IRQK module regards enabling of the resource. Depending on the version or configuration of the Linux kernel, some devices might already be enabled for use and some might not, after booting the kernel. As of this writing IRQK does enable/disable one resource - the MJCP clock. This was added in response to a change in a popular release of the LSP Linux kernel, where the change was to not enable the MJCP clock. This change resulted from a change in philosophy of the kernel - allowing device drivers to enable devices they control, as opposed to having the Linux kernel enable them (or not enable them) without knowledge of their use in the system as a whole. With this approach, if the device driver is enabled then its subject HW is enabled, and if the device driver is not enabled then no power is wasted when enabling an unused device.

IRQK enables the MJCP clock with Linux kernel API clk\_enable(), but only after successfully acquiring it: <syntaxhighlight lang='c'> struct clk \*mjcp\_clk; mjcp\_clk = clk\_get(NULL, "mjcp"); if (IS\_ERR(mjcp\_clk)) {

```
    printk("Error getting mjcp clock\n");
    return -EINVAL;
}
```

clk\_enable(mjcp\_clk); </syntaxhighlight> With the above code, if some device driver was enabled that was also acquiring/enabling the MJCP clock then the above IRQK code will fail, as it should, since some other code is using it. And since this code runs during module insertion into the system, the module will fail to be inserted. The only solution here is to not have that device driver installed when installing irqk.ko.

The code above is #ifdef'd for a particular Linux version. Prior to that version there was no need to enable the MJCP clock since it was not disabled upon booting Linux.

In addition to enabling the MJCP clock during module insertion, IRQK also disables it upon module removal from the system.

Whether or not IRQK needs to enable/disable a resource clock or some other related HW depends on the underlying OS or boot program that is responsible for initializing the system.

VICP User Interface

VICP is the user layer that is responsible for managing resources. It does so through the use of the LinuxUtils module IRQK.

VICP supports management of the following resources:

Supported VICP Resources

```
- VICP_IMX0
- VICP_IMX1
- VICP_MJCP
- VICP_NSF
```

- VICP\_HDVICP0
  - VICP\_HDVICP1

These resources correspond to coprocessor processing elements that can be used independently of each other.

For VICP, resource management means:

- managing ownership
  - managing the resource's interrupt

For interrupt management, VICP supports requesting one of two possible interrupts associated with each resource:

- VICP\_FLEXIBLE
  - VICP\_FIXED

VICP\_FLEXIBLE refers to a general coprocessor interrupt that can be driven by many or all of the VICP resources. When this shared interrupt is granted to one particular resource, any other request for a resource that uses the same interrupt will block until the current resource is released.

VICP\_FIXED refers to a coprocessor interrupt that is driven by only one of the VICP resources. It is possible that such an interrupt will be "switched" by the interrupt multiplexor, requiring the underlying IRQK module to write the appropriate value to the interrupt multiplexor.

The VICP interface also contains a "type" parameter that is used to request that IRQK use one of either the "IRQ" or the "FIQ" interrupt type. As of this writing the LinuxUtils IRQK module does not support specifying this, and is hardcoded to use the "IRQ" type.

## Future LinuxUtils Development

One of the larger goals for LinuxUtils services is to move as much as possible to the user layer, which will help determine the minimum necessary kernel support needed for a particular module. This will ease another goal for LinuxUtils - moving all kernel code into some public Linux GIT release where it can be maintained by the Linux GIT release maintainers.

Also, when possible, implement a LinuxUtils service using the UIO interface. The UIO interface supports user-level code acting in a driver role. IRQK would lend itself nicely to the UIO interface.

### CMEM

As of today's implementation, CMEM requires the system developer to carve out a chunk of physical memory that is not given to the Linux kernel or any other system code (such as DSPLink). This method has some drawbacks, most notably the need to tailor the physical memory to the system in question. To overcome this, CMEM could possibly be implemented to get its memory from the Linux kernel directly, but would need to do so in a way that guarantees that it will get as much contiguous memory as it has been configured to get.

Keystone=

```
{{
1. switchcategory:MultiCore=
  ■ For technical support on
    MultiCore devices, please
    post your questions in the
    C6000 MultiCore Forum
  ■ For questions related to
    the BIOS MultiCore SDK
    (MCSDK), please use the
    BIOS Forum
Please post only comments related
to the article Linux Utils Internals
here.
```

■ For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum

■ For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **Linux Utils Internals** here.

C2000=For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **Linux Utils Internals** here.

DaVinci=For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **Linux Utils Internals** here.


MSP430=For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **Linux Utils Internals** here.

OMAP35x=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Linux Utils Internals** here.

OMAPL1=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Linux Utils Internals** here.

MAVRK=For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **Linux Utils Internals** here.}}

For technical support please post your questions at <http://e2e.ti.com>. Please post on comments about article **Linux U Internals** here.



Amplifiers & Linear

Audio

Broadband RF/IF & Digital Radio

Clocks & Timers

Data Converters

DLP & MEMS

High-Reliability Interface

Logic

Power Management

Processors

■ ARM Processors

■ Digital Signal Processors (DSP)

■ Microcontrollers (MCU)

■ OMAP Applications Processors

Switches & Multiplexers

Temperature Sensors & Control ICs

Wireless Connectivity

Retrieved from "[https://processors.wiki.ti.com/index.php?title=Linux\\_Utils\\_Internals&oldid=182005](https://processors.wiki.ti.com/index.php?title=Linux_Utils_Internals&oldid=182005)"

This page was last edited on 21 July 2014, at 06:52.

Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.