



EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral  
Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

# VERY LARGE TELESCOPE

## INSTRUMENTATION DIVISION




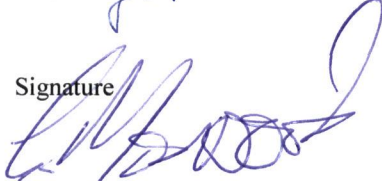
### New General detector Controller

#### NGC-LCU Interface Software – User Manual

Document Number: VLT-MAN-ESO-13660-4560

Document Issue: <1.0>

Date of Issue: <30/06/2008>

Prepared by :	Name	Date	Signature
	Jörg Stegmeier	30.6.2008	
Approved by :	Name	Date	Signature
	Dietrich Baade, Gert Finger	7.7.2008 30.6.2008	 
Released by :	Name	Date	Signature
	Alan Moorwod	07/07/08	



### CHANGE RECORD

ISSUE	DATE	SECTIONS AFFECTED	REASON/INITIATION DOCUMENTS/REMARKS
0.1	16-05-2008	All	First draft
1.0	30-06-2008	8	Added section for system test.



## Table of Contents

1. Introduction.....	6
1.1. Purpose .....	6
1.2. Scope.....	6
1.3. Applicable Documents.....	7
1.4. Reference Documents .....	7
1.5. Abbreviations and Acronyms .....	7
1.6. Glossary .....	7
1.7. Stylistic Conventions .....	7
1.8. Naming Conventions .....	8
1.9. Problem Reporting/Change Request.....	8
2. Overview.....	9
2.1. System Architecture.....	9
2.2. PMC Interface .....	10
2.3. Specifications .....	10
2.4. Hardware and Software Requirements .....	11
2.5. NGC System Configuration .....	11
2.6. Test Results .....	12
3. Installation .....	13
4. Device Driver .....	14
4.1. Driver and Device Installation .....	14
4.2. Device Access Functions.....	15
4.2.1. open .....	15
4.2.2. close.....	15
4.2.3. ioctl.....	15
4.3. General Tools .....	16
5. Driver IOCTL Commands.....	17
6. Data Capture .....	20
7. Task Handling.....	21
7.1. Task Registration .....	21
7.2. Run Control .....	22
7.3. Parameter Access .....	22
8. System Test.....	24
8.1. After Power-Up.....	24
8.2. Interactive Communication .....	24
8.3. Visualization.....	26
8.4. Verification.....	26
9. Reference .....	27
9.1. Error Definitions .....	27
9.2. ngclcuDrv(3) .....	29
9.3. ngclcuDevCreate(3).....	30
9.4. ngclcuIoctl(3) .....	31



9.5. ngclcuInterrupt(3) .....	34
9.6. ngclcuDevice(3) .....	35
9.7. ngclcuTools(3) .....	37
9.8. ngclcuCapture(3).....	39
9.9. ngclcuTask(3).....	41
9.10. ngclcuServer(1).....	44



## List of Figures

Figure 1 System Architecture .....	9
Figure 2 PMC Interface.....	10

## List of Tables

Table 1 Error Definitions .....	28
---------------------------------	----



## 1. Introduction

The software described in this manual is intended to be used in the ESO VLT project by ESO and authorized external contractors only.

While every precaution has been taken in the development of the software and in the preparation of this documentation, ESO assumes no responsibility for errors or omissions, or for damage resulting from the use of the software or of the information contained herein.

### 1.1. Purpose

This document is the user manual of the LCU interface software for the New General detector Controller (NGC).

Its purpose is to provide people, who intend to process the NGC video-data on an ESO standard LCU running the VxWorks operating system, with all the necessary information to install the *ngclcu* software module and to build their own data-processing applications on top of it. The *ngclcu* software module contains a device driver to access the NGC-PMC interface device (see [RD1]) as well as some driver interface functions for data capture and acquisition task handling.

Main applications for this NGC-LCU interface are the fast control loops with low latency required by the VLTI instruments. It has to be distinguished clearly from the interface to the SPARTA system for adaptive optics applications [RD11].

The manual assumes that the reader has some knowledge of the C/C++ programming language, UNIX and VxWorks Operating Systems, and the VLT Software, in particular CCS. It is not intended to be an introduction to LCU installation/configuration and therefore it uses common terminology in this field without further explanation (e.g. *bootScript*, *userScript*, *CCS environment*, etc.).

A conscious effort has been made to maintain a certain degree of backwards compatibility of the *ngclcu* software module with the *irvme* software module [RD5], which is the counterpart for the IRACE controller [RD4]. Nevertheless no responsibility is assumed if this goal is missed in some areas. Where applicable a hint to the major changes with respect to *irvme* can be found at the end of each section.

### 1.2. Scope

Scope of this document is the *ngclcu* software module. The interface hardware is described in [RD1]. The software module is under CMM configuration control.



### 1.3. Applicable Documents

The following documents, of the exact issue shown, form a part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this document, the contents of this document shall be considered as a superseding requirement.

[AD1]	VLT-SPE-ESO-13660-3207, 1.0	NGC Requirements Specification
[AD2]	VLT-SPE-ESO-13660-3670, 1.0	NGC Software Requirements
[AD3]	VLT-LIS-ESO-13660-3907, 1.0	NGC Project Glossary
[AD4]	VLT-LIS-ESO-13660-3908, 1.0	NGC Project Acronyms
[AD5]	VLT-PRO-ESO-10000-0228, 1.0	VLT Software Programming Standards
[AD6]	VLT-MAN-ESO-17210-0667, 1.3	Guidelines for Development of VLT Application Software
[AD7]	VLT-SPE-ESO-17212-0001, 5.0	VLT Instrumentation Software Specification
[AD8]	VLT-SPE-ESO-17240-0385, 4.0	INS Common Software Specification
[AD9]	GEN-SPE-ESO-19400-0794, 3.0	Data Interface Control Document
[AD10]	VLT-MAN-ESO-17200-0642, 5.0	VLT Common Software – Installation Manual
[AD11]	VLT-MAN-ESO-17240-2325, 5.0	INS Common Software – Configuration Tool – User Manual
[AD12]	VLT-MAN_ESO-17210-0375, 2.2	VLT Software CCS-LCU Driver Development Guide and User Manual

### 1.4. Reference Documents

The following documents are referenced in this document.

[RD1]	VLT-MAN-ESO-13660-4510	New General Detector Controller (NGC) - User Manual
[RD2]	VLT-MAN-ESO-13660-4085	NGC Infrared DCS - User Manual
[RD3]	VLT-MAN-ESO-13660-4086	NGC Optical DCS - User Manual
[RD4]	VLT-MAN-ESO-14100-1878	VLT Software - IRACE-DCS - User Manual
[RD5]	VLT-MAN-ESO-14100-2457	VLT Software - IRACE VME-BUS Interface Driver - User Manual
[RD6]	VLT-MAN-ESO-17210-1358	VLT Software - CCS-LCU Configuration of VLT Standard VME Boards
[RD7]	VLT-MAN-ESO-17210-0690	VLT Software - Graphical User Interface - User Manual
[RD8]	VLT-MAN-ESO-17240-0866	Real Time Display - User Manual
[RD9]	VLT-MAN-ESO-17200-0908	Tools for Automated Testing - User Manual
[RD10]	GEN-SPE-ESO-00000-0949	VLT Time Reference System Time
[RD11]	VLT-SPE-ESO-16100-3729	SPARTA - Adaptive Optics Real Time Computer Platform - Spec. for NGC

### 1.5. Abbreviations and Acronyms

Abbreviations and acronyms used in the NGC project are listed in [AD4].

### 1.6. Glossary

All the relevant concepts used within the NGC project are listed in [AD3].

### 1.7. Stylistic Conventions

The following styles are used:

**bold** in the text, for commands, file names, etc. as they must be typed.

*italic* in the text, for parts that have to be substituted with the real content before typing.

`courier` for examples.



<name> in the examples, for parts that have to be substituted with the real content before typing.

The **bold** and *italic* styles are also used to highlight words.

## 1.8. Naming Conventions

This implementation follows the naming conventions as outlined in [AD7].

## 1.9. Problem Reporting/Change Request

The form described in [AD10] shall be used.



## 2. Overview

### 2.1. System Architecture

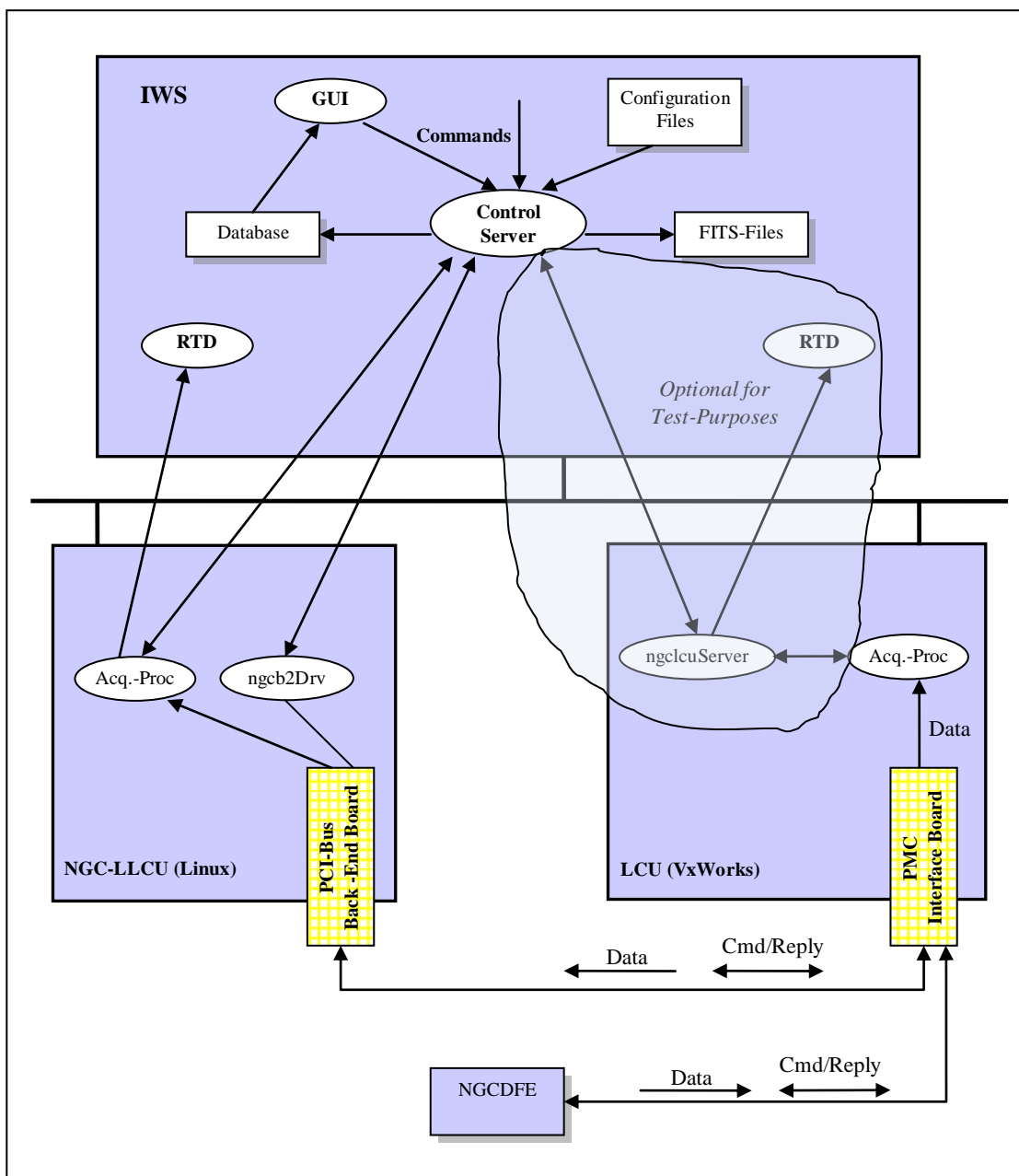


Figure 1 System Architecture

## 2.2. PMC Interface

The PMC interface is a 64-bit DMA master interface to a PCI-bus in PMC format (see Figure 2). Video data packets from the downstream link are recognized and written to the video data FIFO. From there DMA transfers to the PMC PCI-bus can be executed. Additionally the video data packets are forwarded to the upstream link. Packets containing commands or replies are just routed through and are not seen from the PCI-bus. A more detailed description of the hardware is given in [RD1].

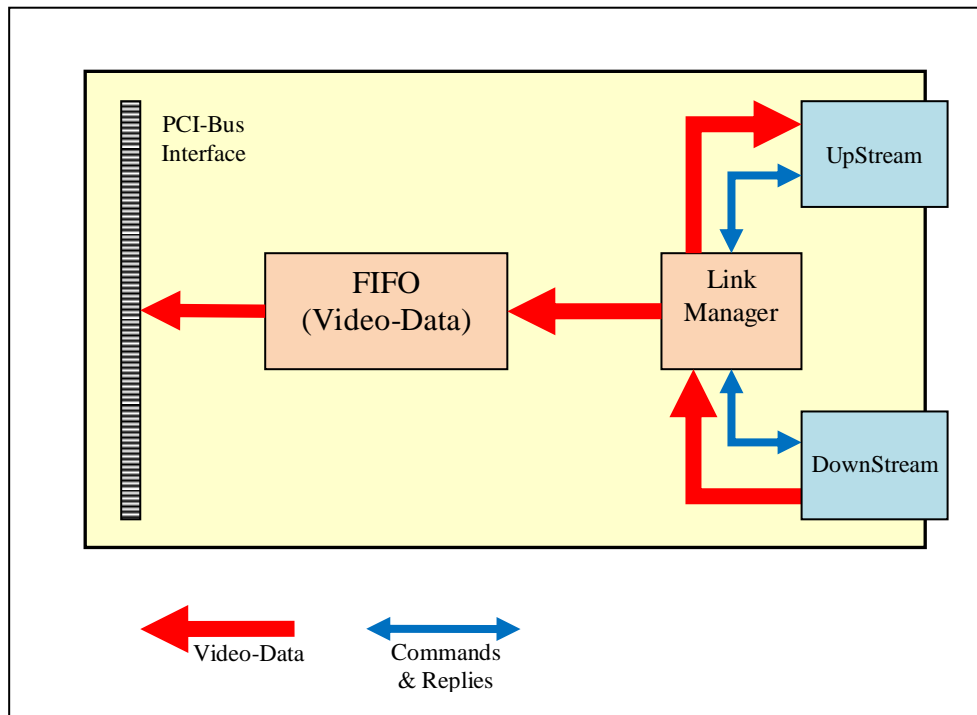


Figure 2 PMC Interface

## 2.3. Specifications

PCI Interface:	64 Bit / 33 MHz
Minimum DMA Block-Size:	32 bytes (16 pixels)
Maximum DMA Block-Size:	512 bytes (256 pixels)
Maximum DMA-Size:	Max. Phys.-Mem avail. / 2



## 2.4. Hardware and Software Requirements

To use the *ngclcu* software module the following hardware and software environment is required (see also [RD6]):

- 1 NGC Detector Front-End.
- 1 NGC-LLCU (Linux PC) for NGC command and configuration control.
- 1 VME-BUS chassis with P1/J1 and P2/J2 bus backplanes and power supplies.
- 1 Motorola MVME6100 CPU board.
- 1 NGC-PMC Interface Board (installed in PMC slot-1 of the MVME6100)
- VLTSW release VLT2008 or later.
- VxWorks version 5.5 operating system or higher.
- Software module *lculog* for internal logging, version 1.12 or higher.
- Software module *lcudrv* for common driver function, version 1.35 or higher.
- LCU software environment to boot from.

## 2.5. NGC System Configuration

The PMC interface always strips off one header from each link packet and therefore has an influence on the routing tables (see [RD2]) of all NGC DFE modules (sequencer, CLDC, ADC) connected behind. When the PMC interface is installed all routes have to be prefixed with an additional link (5). E.g. the CLDC-module on the first basic board would get:

```
DET.CLDC1.ROUTE    "5,2";    # route to module (with PMC)
```

Instead of:

```
DET.CLDC1.ROUTE    "2";      # route to module (without PMC)
```

The ADC-module on a second board would get:

```
DET.ADC2.ROUTE     "5,5,2"; # route to module (with PMC)
```

Instead of:

```
DET.ADC2.ROUTE     "5,2";    # route to module (without PMC)
```

For large systems such modifications are a bit inconvenient. Therefore the *NGIRSW* can be instructed to add the prefix internally and hide these configuration issues from the user. The keyword *DET.DEVi.PMC* “*T|F*” has to be used for that purpose. The system will then automatically adjust the routing tables accordingly.



## 2.6. Test Results

Measured latency with the MVME6100 CPU:

$$\begin{aligned} 4\mu\text{s} + n \cdot (0.004 \mu\text{s}) & \quad (n < \text{DMA } \textit{BlockSize}) \\ 4\mu\text{s} + \textit{BlockSize} \cdot (0.004 \mu\text{s}) & \quad (n \geq \text{DMA } \textit{BlockSize}) \end{aligned}$$

where  $n$  is the number of bytes to be transferred. Using a 512 bytes DMA block-size a maximum latency of 6  $\mu\text{s}$  has been measured.

This latency adds to the read-out time. The latency of any VxWorks internal task synchronization mechanism (e.g. semaphores) is not considered. So this is the minimum latency where the applications code is executed at interrupt level (see section 5).



### 3. Installation

Use the following instructions to retrieve and install the *ngclcu* software module:

- `cmmCopy ngclcu`
- `export CPU=PPC604`
- `cd ngclcu/src/`
- `make man all install`
- `cd ngclcu/test`
- `make man all install`

To have the *ngclcu* module installed (see section 4.1) at LCU boot-time the following has to be added to the *userScript* of the LCU environment:

```
lcubootAutoLoadNoAbort 1,"ngclcu",0  
lcubootAutoCdBoot 1,"ngclcu.boot"  
< ngclcu.boot
```

This installs the basic version without task-interface, data-server and test-facilities (i.e. only the driver interface as described in sections 4, 5 and 6 can be used). In order to install the full version at LCU boot-time the following has to be added to the *userScript* of the LCU environment:

```
lcubootAutoLoadNoAbort 1,"ngclcu",0  
lcubootAutoCdBoot 1,"ngclcu_all.boot"  
< ngclcu_all.boot
```



## 4. Device Driver

### 4.1. Driver and Device Installation

The installation of the *ngclcu* driver is according to the VxWorks driver concept. Two functions are provided:

*ngclcuDrv* installs the driver to the VxWorks I/O system. It takes the arguments:

- a) The maximum number of supported devices.
- b) The maximum number of channels that can be opened (should be at least 2).
- c) The device access timeout in ticks (typically a value of 100 is used).

*ngclcuDevCreate* creates a device. It takes the arguments:

- a) Device name (typically “/ngc0”).
- b) The base address of the device on the PCI-Bus. The value 0x0 will use an automatic configuration (recommended). The value 0xffffffff will create the device in simulation mode. In simulation mode the devices registers are mapped in the CPU RAM and data interrupts are generated via a timer. The simulation interrupt interval can be set via the *ngclcuCMD\_SIMTIME ioctl*-command. The timer resolution is based on the VxWorks system clock (currently this is 10 ms with VxWorks 5.5 and MVME6100).



## 4.2. Device Access Functions

The *ngclcu* driver supports the functions *open*, *close* and *ioctl*. The driver calls are mutually exclusive. The device accessed by a task is protected from being accessed from any other authorized task until the executing driver call has terminated. A task blocked for access has a timeout. An error will be returned upon exceeding the timeout.

### 4.2.1. open

A channel to an *ngclcu* device can be opened with the *open* system call either in read-only mode (*lcudrvOPEN\_READONLY*) or shared mode (*lcudrvOPEN\_SHARED*). The device name is typically “/ngc0”:

```
channel = open("/ngc0", lcudrvOPEN_SHARED, (int)&status);
```

If the *open* call fails a negative value is returned. The status argument provides the specific error reason. An appropriate error message is returned by the *ngclcuErrorGet(status)* function. If the operation was successful the status is set to *lcudrvOK*.

### 4.2.2. close

The *close* system call is used to close a channel to an *ngclcu* device:

```
status = close(channel);
```

The function returns *lcudrvOK* in case of successful operation. Otherwise an appropriate error message is returned by the *ngclcuErrorGet(status)* function.

### 4.2.3. ioctl

The *ioctl* system call is used to send a command to an *ngclcu* device:

```
status = ioctl(channel, command, (int)&argument);
```

The function returns *lcudrvOK* in case of successful operation. Otherwise an appropriate error message is returned by the *ngclcuErrorGet(status)* function. The command is a number identifying the operation to be performed by the driver. Literals of all commands are defined in *ngclcuCommands.h*. The valid *ioctl* commands are described in section 3. The argument is the address of the command argument or *NULL* if no argument is used. All argument data structures are defined in *ngclcuCommands.h*.

#### Changes with respect to irvme:

*ngclcu* implements the same device access functions as *irvme*.



### 4.3. General Tools

The following tool functions are provided:

- ***ngclcuVersion()*** shows the version of the loaded driver module on the LCU-console.
- ***ngclcuDevShow()*** prints a list of all installed NGC-PMC interface devices with their respective installation parameters to the LCU console.
- ***ngclcuMalloc()*** allocates a cache-aligned buffer for DMA transfers. The syntax is the same as for the standard ***malloc()*** system call.
- ***ngclcuFree()*** frees a buffer that has been allocated with the ***ngclcuMalloc()*** function.
- ***ngclcuErrorGet()*** returns a pointer to a string matching the given status.
- ***ngclcuTransferStat2Str()*** returns a pointer to a string matching the given data transfer status.
- ***ngclcuStatus()*** prints the current device status on stdout. The device is specified by its instance number.
- ***ngclcuClr()*** resets the FIFOs on a device specified by its instance.
- ***ngclcuReadReg()*** reads a register value from a device specified by its instance.
- ***ngclcuWriteReg()*** writes a value to a register on a device specified by its instance.
- ***ngclcuIntOff()*** disables the interrupts from a device specified by its instance.
- ***ngclcuIntOn()*** enables the interrupts from a device specified by its instance.



## 5. Driver IOCTL Commands

The following *ioctl* commands are supported by the *ngclcu* driver. The command literals are defined in *ngclcuCommands.h*.

- *ngclcuCMD\_FREE\_DEVICE* frees a device;  
The argument is ignored;
- *ngclcuCMD\_RESET\_DEVICE* resets a device (data-FIFO is cleared);  
The argument is ignored;
- *ngclcuCMD\_READ\_REG* reads a device register;  
The argument is a pointer to an *ngclcuREG* object:

```
UINT32 reg;      - Register (offset)
UINT32 value;    - Value
```

- *ngclcuCMD\_WRITE\_REG* writes to a device register;  
The argument is a pointer to an *ngclcuREG* object.

```
UINT32 reg;      - Register (offset)
UINT32 value;    - Value
```

- *ngclcuCMD\_READ\_PCI* reads from a PCI configuration space register;  
The argument is a pointer to an *ngclcuREG* object.

```
UINT32 reg;      - Register (offset)
UINT32 value;    - Value
```

- *ngclcuCMD\_WRITE\_PCI* writes to a PCI configuration space register;  
The argument is a pointer to an *ngclcuREG* object.

```
UINT32 reg;      - Register (offset)
UINT32 value;    - Value
```

- *ngclcuCMD\_ACK\_INT* acknowledges that the device interrupt is handled;  
The argument is ignored;

- ***ngclcuCMD\_ATTACH\_INT*** attaches to the device data interrupt;  
The argument is a pointer to an ***ngclcuCFG\_INT*** object:

```
SEM_ID semISR;           - Interrupt service semaphore
ngclcuUSR_ISR *userISR; - User routine at interrupt level
void *userArg;          - Pointer to user argument
BOOL trigger;           - Trigger task on interrupt
```

If ***userISR*** is not a ***NULL*** pointer the given user routine is called at interrupt level. The user routine is called with three arguments: a pointer to the received data buffer, the size of the buffer and the given ***userArg***. If ***trigger*** is set to ***TRUE*** the returned semaphore ***semISR*** can be used to additionally trigger a routine at task level. If ***userISR*** is a ***NULL*** pointer, ***semISR*** is always given in the interrupt service routine regardless of the value of ***trigger***. The user routine is defined in ***ngclcuCommands.h*** as follows:

```
typedef void (ngclcuUSR_ISR)(void *buffer, int size, void *userArg);
```

- ***ngclcuCMD\_DETACH\_INT*** detaches from the device data interrupt;  
The argument is ignored.
- ***ngclcuCMD\_INT\_ENABLE*** enables interrupt generation;  
The argument is ignored.
- ***ngclcuCMD\_INT\_DISABLE*** disables interrupt generation;  
The argument is ignored.
- ***ngclcuCMD\_STATUS*** gets the i/o-status;  
The argument returns a pointer to an ***ngclcuSTATUS*** object:

```
int pIdx;           - Buffer index for process
int tstat;          - Transfer status - this is one of:

    ngclcuTSTAT_OK           - Successful transfer
    ngclcuTSTAT_OVERFLOW    - FIFO overflow
    ngclcuTSTAT_ERROR        - Link i/o-error

BOOL overrun;      - Buffer-overflow flag
BOOL attached;     - Attached to interrupt
```

- ***ngclcuCMD\_HWSTATUS*** gets the device hardware status;  
The argument returns a pointer to an ***ngclcuHWSTATUS*** object:

```
UINT32 hwStatus;    - Content of the device status register

D0-4:  Reserved
D5:    FIFO empty
D6:    FIFO full
D7:    Overflow flag (cleared with ngclcuCMD_RESET)
D8:    Upstream link-channel up
D9-11: Upstream hard-/soft-/framing-error
D12:   Downstream link-channel up
D13-15: Downstream hard-/soft-/framing-error
D16-31: Reserved

UINT32 revision;   - Content of the device revision register

D0-3:   Board type (9 = PMC interface)
D4-7:   Sub-type (1)
D8-11:  Hardware revision number
D12-15: Firmware revision number
D16-19: Firmware sub-revision number
D20-31: Reserved

int dmaSize;      - DMA block-size
```

- ***ngclcuCMD\_CONFIG*** configures the transfer;  
The argument is a pointer to an ***ngclcuCONFIG*** object:

```
void *data[2];    - Pointers to user double buffer
int d_size;       - Buffer size in bytes
int b_size;       - DMA block-size in bytes (zero = default)
```

- ***ngclcuCMD\_SIMTIME*** sets the simulation interval;  
The argument is a pointer to an ***ngclcuSIMTIME*** object:

```
int simTime;     - Simulation interval in microseconds
```

### **Changes with respect to irvme:**

*The ***ngclcuCMD\_READ/WRITE\_REG*** and ***ngclcuCMD\_READ/WRITE\_PCI*** commands are new. The ***ngclcuCMD\_HWSTATUS*** command has changed to reflect the new hardware. The ***ngclcuCMD\_CONFIG*** takes the DMA block-size as additional parameter. This can be set to zero to let the system choose (as the *irvme* always does). The “error” parameter in the ***ngclcuCMD\_STATUS*** command has been removed as no error interrupt is generated by the NGC PMC interface. The other commands and parameters are backwards compatible with the *irvme* module.*



## 6. Data Capture

For easier data access several data capture control functions have been implemented on top of the *ioctl*-calls. To use these function the *ngclcuCapture.h* header file has to be included.

- *ngclcuInit()* returns a channel to a device specified by *devName*. If the channel could not be opened (-1) is returned. The *devName* typically is “/ngc0”.
- *ngclcuReset()* resets the device. FIFO and FIFO-full-flag are cleared, but all other configuration is kept.
- *ngclcuConfig()* configures a data capture via a device specified by a channel returned by *ngclcuInit()*. The (double-)buffer passed to this function has to be allocated by the calling routine. A *NULL* pointer indicates that the previous buffers should be re-used. If no buffers have been configured with a prior call an error is returned. The size of the buffer and the DMA block-size are given in bytes. A zero block-size tells the system to use a default value.
- *ngclcuStart()* starts the data capture loop and returns a semaphore to be passed to following *ngclcuWait()* calls. A user defined routine can be specified which is called at interrupt level with three arguments: a pointer to the received data buffer, the size of the buffer and the given user argument.
- *ngclcuStop()* aborts the data capture loop (asynchronous call).
- *ngclcuWait()* waits with timeout (in ticks) for the next data buffer and returns the buffer index that has to be used for the processing. If *WAIT\_FOREVER* is used as timeout the routine waits until the transfer has completed or an error/abort has occurred. The function returns a negative value (error code) if an error occurred. If the transfer has been aborted *ngclcuERROR\_ABORTED* is returned as error code, which has to be caught by the calling application.
- *ngclcuAck()* has to be called when all processing is done with the current buffer.
- *ngclcuRelease()* should be called to release (and close) a channel to a device.

### Changes with respect to irvme:

*The data capture functions are backwards compatible with the irvme module except ngclcuConfig() which has the DMA block-size as additional argument. Setting the block-size to zero will provide the same behavior as the irvme module (i.e. let the system choose an appropriate value).*



## 7. Task Handling

The real-time applications can be controlled via the standard *NGC-DCS* in the same way as a normal data acquisition process (see [RD2]). For that purpose the *ngclcuServer* task has to be launched on the VxWorks platform, which runs the same TCP/IP socket protocol as the standard NGC acquisition processes. The command port number is given as parameter to the *ngclcuServer* task. The data port number defaults to command port number plus one. It can also be given as second argument to *ngclcuServer()*.

The server can execute any application specific data capture task that has previously been registered. Run-control and parameter definitions are done in a similar way as with the standard acquisition process. There also exists a *NGC-DCS* compatible data interface, but here only one frame type (SAMPLE-Frame) is supported, as the *ngclcu* module has been designed for real-time control loops rather than for imaging exposures. When specifying a real-time task in the detector configuration, the task name (as given at registration) must be prefixed by an “*rt\_<name>*” token.

### 7.1. Task Registration

When installing an application module all exported tasks (i.e. tasks which should be started and stopped by the *ngclcuServer*) must first be registered by calling the *ngclcuTaskRegister* function:

```
int ngclcuTaskRegister( const char *name,  
                        FUNCPTR entryPt,  
                        int options,  
                        int stackSize,  
                        int numParam,  
                        ngclcuPARAM *param,  
                        int protectionTimeout,  
                        char *erms )
```

The synopsis is similar to the VxWorks *taskSpawn* function. Additionally an array of parameter records has to be passed. These records define name and type and also a default value for each parameter that should be set via the dynamic parameter facility of the *NGC-DCS*:

```
char name[64]; - parameter name  
int value;    - parameter (default) value  
int type;    - parameter type
```

Supported parameter types are *ngclcuPTYPE\_INT* (32 bit integer) and *ngclcuPTYPE\_FLOAT* (32 bit floating point). The protection timeout for the parameters is specified in ticks. An internal task-id is returned which is also passed to the task as (only) parameter. This id has to be used for all other calls to the *ngclcu* task interface functions. The task is spawned, when the server receives a start command.



## 7.2. Run Control

The task execution is synchronized to the start-/stop-command through calls to *ngclcuTaskStart(id)/ngclcuTaskCont(id, brk)*. If these functions return a **FALSE** value the task should terminate. By setting the brk-flag to **FALSE** the task signals to the server, that it will not exit at the stop-command, but will wait again for the next start-command. If the brk-flag is set to TRUE, the task should terminate immediately.

Additionally two external flags are supported for run-control. The function *ngclcuTaskFlags(id, &resetFlag, &endFlag)* returns the values of the two flags and internally clears them after each call. If the *resetFlag* is set, the acquisition loop should reset the current exposure and start from the beginning. If the *endFlag* is set, the acquisition loop should end the exposure as soon as possible (for example by transferring an intermediate result).

All fatal errors (i.e. errors which require that the task must be stopped and restarted, or which force the task to terminate) should be reported through the *ngclcuErrStackAdd(msg)* function.

## 7.3. Parameter Access

The parameters are accessed through the *ngclcuTaskParam(id)* function. This function returns a pointer to a task specific parameter structure. The pointer should be casted to a structure with integer/floating-point values in the same order as given in the *param* argument to *ngclcuTaskRegister()*. The function can be called at any time and returns the actual parameter setup values. When the task is selected via the **NGC-DCS** an argument string is passed containing some configuration parameters, which are not subject to changes during run-time. A pointer to a **NULL** terminated list of these arguments is returned by the function *ngclcuTaskArgGet()*. Generally all tasks producing frames which have to be transferred to the **NGC-DCS** should take the **-nx**, **-ny** arguments as output format.

It is possible to access data-sets like flat-fields and bad-pixel-masks, which are downloaded via the *ngclcuServer*. This is handled via the *ngclcuTaskMem(memId, size, &valid)* function. The function returns a pointer to data-set memory identified by *memId*. The *memId* has to be a single-bit value. The id-assignment is the responsibility of the task application(s).

The first call to *ngclcuTaskMem()* to allocate a buffer for a new data-set should be done during the registration phase (to let the *ngclcuServer* load the data-sets before the tasks are started). If *size* is greater than zero, *size* bytes of memory are allocated. If *size* is equal to **ngclcuTASK\_MEM\_FREE**, the data-set is removed. A **NULL** pointer is returned in this case. If *memId* is equal to **ngclcuTASK\_MEM\_ALL**, all data-sets are removed (in this case *valid* can be specified as a **NULL** pointer). If the function is called again with



the same *memId* but different size, then the memory of the data-set is reallocated and marked as invalid. If the requested memory could not be allocated a *NULL* pointer is returned. If *size* is equal to *ngclcuTASK\_MEM\_NO\_CHANGE* only the buffer belonging to the specified *memId* (or a *NULL* pointer if not yet initialized) is returned. If the memory of the data-set has been validated by the server or by an explicit call to *ngclcuTaskMemValidate(memId)* the *valid* flag is set to *TRUE*. A task application must never use the memory of a data-set marked as invalid.

**Changes with respect to irvme:**

*The task-handling is backwards compatible with the irvme module.*



## 8. System Test

*Attention:* This introduction is also available in ASCII format in the **ngclcu** SW-module (`ngclcu/doc/README`).

### 8.1. After Power-Up

- Wait until the LCU responds to PING requests “`ping <hostname>`”.
- Then try “`rlogin <hostname>`” until the shell is successfully opened. You will receive a message like “*Sorry, the shell is locked.*” until the LCU has executed all instructions in the “`bootScript`”.
- Then either type “`exit`” to exit the shell or leave it opened to see the log-messages issued by the **ngclcu** software.

### 8.2. Interactive Communication

A tool is available within the NGC Base-SW (module **ngcpp**, see [RD2]) which allows the interactive communication with the **ngclcu** software:

```
ngcppShell -host <hostname> -port 9000
```

The following commands are supported:

#### **connect**

Reconnect the shell after LCU reboot. I.e. the **ngcppShell** can be left running and the command stack is saved.

#### **exit**

Exit the **ngcppShell**.

```
iow config <address (HEX)> <value (HEX)>
```

Write a value to an address in the PCI-configuration space.

```
ior config <address (HEX)>
```

Read a value from an address in the PCI-configuration space.

```
iow local <address (HEX)> <value (HEX)>
```

Write a value to an address in the local address space.

```
ior local <address (HEX)>
```

Read a value from an address in the local address space.





***rmode rt\_<taskName> <options>***

Launch a new processing task with name *<taskName>* on the LCU. The *<options>* are task specific. Typically the options *-nx <pixels>* and *-ny <pixels>* are supported to communicate the image size.

Example:

```
rmode rt_taskAdc -nx 256 -ny 256
```

The following tasks are currently available:

- *taskSimple [-nx <n>] [-ny <n>] [-dmasize <bytes>]*

Default values (if not specified):

```
nx = 256  
ny = 256  
dmasize = 0 (let system choose)
```

- *taskAdc [-nx <n>] [-ny <n>] [-nadc <n>] [-dmasize <bytes>]*

Default values (if not specified):

```
nx = 256  
ny = 256  
nadc = 4  
dmasize = 0 (let system choose)
```

Other tasks will be implemented upon request.

***start***

Start sustained DMA data acquisition.

***stop***

Stop sustained DMA data acquisition.

***Remark:*** The other commands of the *ngcppShell* tool (see “help” command) are supported as well, but do not have further meanings within the context of the procedure described in this section.



### 8.3. Visualization

The data can be visualized using the standard *ngcrtd* tool (see [RD2]):

```
ngcrtd -host <hostname> -dataport 9001 [-appname PMC -camera PMC]
```

The *-appname* and *-camera* options are only needed when another *ngcrtd* instance is already running. Other names than “*PMC*” may be used (e.g. “*NGC*”, “*MYRTD*”, etc.).

**Caution:** *The ngcrtd has to be re-launched after LCU-reboot!*

### 8.4. Verification

The data can printed out in HEX-byte format and/or dumped to a file:

```
ngcppDart -host <hostname> -port 9001 [-v <2|3>] [-o <fileName>]
```

The *-v* options specifies a verbose level. In level 2 only the first bytes printed to the screen. In level 3 all bytes are printed.

The *-o* option specifies a filename where the data values are dumped. If the file extension is “*.bin*” the data is written in raw binary format. If the file extension is “*.dat*” the data is written in ASCII format (20 bytes per row in HEX-byte format separated by white space). If no extension is given “*.bin*” is assumed by default. The files will get a sequential number for each received frame (i.e. “*myFile\_0001.bin*”, “*myFile\_0002.bin*”, etc.).

To verify data integrity one *ngcppDart* process for the LCU and one for the NGC-LLCU (Linux-PC) has to be launched:

```
ngcppDart -host <Linux-PC> -o <file>_linux.bin (or .dat)
```

```
ngcppDart -host <LCU> -port 9001 -o <file>_lcu.bin (or .dat)
```

After starting the system the produced files can be compared:

```
diff <file>_linux_0001.bin <file>_lcu_0001.bin
```

or

```
diff <file>_linux_0001.dat <file>_lcu_0001.dat
```

The files must be identical.



## 9. Reference

### 9.1. Error Definitions

In addition to the errors defined by the *lcudrv* module the following error-codes can be returned by the installation functions (*ngclcuDrv*, *ngclcuDevCreate*) and the driver calls (*open*, *close*, *ioctl*):

<u>Error</u>	<u>Code</u>	<u>Description</u>
<b>ngclcuERROR_OPEN_CHANNELS</b>	-201	The operation could not be performed as still open channels exist.
<b>ngclcuERROR_INT_NOT_VALID</b>	-211	The content of the device interrupt register is invalid.
<b>ngclcuERROR_INT_NO_HANDLER</b>	-212	The interrupt handler for the specified interrupt number and level could not be built.
<b>ngclcuERROR_SEMAPHORE_IVLD</b>	-213	It has been tried to access an invalid semaphore. The semaphore was either not created successfully or has already been deleted.
<b>ngclcuERROR_ATTACHED</b>	-221	The operation could not be performed as the driver is already attached to the device interrupts.
<b>ngclcuERROR_DETACHED</b>	-222	The operation could not be performed as the driver is not yet attached to the device interrupts.
<b>ngclcuERROR_CFG_NO_CFG</b>	-231	The operation could not be performed as the data transfer has not yet been configured.
<b>ngclcuERROR_CFG_IVLD_DSIZE</b>	-232	An invalid size for the user data buffers has been specified.



<u>Error</u>	<u>Code</u>	<u>Description</u>
<b>ngclcuERROR_CFG_IVLD_PTR</b>	-233	An invalid pointer to a user data buffer has been specified.
<b>ngclcuERROR_DMA_INIT</b>	-241	The DMA initialization failed.
<b>ngclcuERROR_ABORTED</b>	-251	The transfer has been aborted (via stop command).
<b>ngclcuERROR_TRANSFER</b>	-252	An error occurred during data transfer. The transfer status can be retrieved with the <i>ngclcuCMD_STATUS ioctl</i> -command.
<b>ngclcuERROR_PCI_NOT_FOUND</b>	-261	The device was not found on the PCI-Bus.
<b>ngclcuERROR_PCI_MEM_MAP</b>	-262	The PCI-bus memory mapping for the device failed.

**Table 1 Error Definitions**



## 9.2. ngclcuDrv(3)

### NAME

ngclcuDrv - Install the NGCLCU driver module

### SYNOPSIS

```
#include "ngclcu.h"
```

```
int ngclcuDrv(int devices, int channels, int timeout);
```

```
int ngclcuDrvRemove(void);
```

### DESCRIPTION

ngclcuDrv() is called only once at startup. It hooks up the various I/O service calls to the driver's functions, assigns the driver number, and adds the driver to the driver table.

devices - number of supported devices  
channels - number of channels that can be simultaneously be opened  
timeout - access timeout value

ngclcuDrvRemove() removes an NGCLCU driver.

### RETURN VALUES

lcudrvOK - driver successfully installed  
lcudrvERROR\_DRIVER\_EXISTS - the driver is already installed  
lcudrvERROR\_NO\_MEMORY - there is not enough memory for dynamic data structures  
lcudrvERROR\_INVALID\_ARGUMENT - the value of one of the parameters is out of range  
ngclcuERROR\_OPEN\_CHANNELS - open channels exist

### SEE ALSO

ngclcuDevCreate(3), ngclcuInterrupt(3), ngclcuOpen(3), ngclcuClose(3), ngclcuIoctl(3), ngclcuDevice(3), ngclcuTools(3), open(2), close(2), ioctl(2)



## 9.3. ngclcuDevCreate(3)

### NAME

ngclcuDevCreate - Add an I/O-device to the NGCLCU driver

### SYNOPSIS

```
#include "ngclcu.h"
```

```
int ngclcuDevCreate(char *devName, void *baseAddr);
```

```
int ngclcuDevDelete(int unit);
```

### DESCRIPTION

ngclcuDevCreate() is called at startup for each device to be installed. It adds a device to the driver, making it available for subsequent open operations.

- devName - device name (must be "/ngcX" X>=0, e.g. "/ngc0")
- baseAddr - PCI base address of the board in the PCI address space. The value 0x0 will use an auto-configuration (recommended). The value 0xffffffff will create the device in simulation mode.

ngclcuDevDelete() deletes a device specified by its unit.

### RETURN VALUES

- lcudrvOK - successful completion
- lcudrvERROR - there is a general VxWorks I/O error
- lcudrvERROR\_NO\_DRIVER - driver not yet installed
- lcudrvERROR\_INVALID\_DEVICE - invalid device name
- lcudrvERROR\_INVALID\_ARGUMENT - invalid parameter
- lcudrvERROR\_DEVICE\_EXISTS - device already installed
- lcudrvERROR\_NO\_SEMAPHORE - creation of access protection semaphore failed
- ngclcuERROR\_OPEN\_CHANNELS - open channels exist
- ngclcuERROR\_SEMAPHORE\_IVLD - tried to access invalid semaphore
- ngclcuERROR\_INT\_NO\_HANDLER - no interrupt handler
- ngclcuERROR\_PCI\_NOT\_FOUND - device not found on PCI-Bus
- ngclcuERROR\_PCI\_MEM\_MAP - PCI memory mapping failed

### EXAMPLE

```
int status;  
status = ngclcuDevCreate("/ngc0", 0x0);  
if (status != lcudrvOK)  
{  
    // error process  
}
```

### SEE ALSO

ngclcuDrv(3), ngclcuInterrupt(3), ngclcuOpen(3), ngclcuClose(3),  
ngclcuIoctl(3), ngclcuDevice(3), ngclcuTools(3), open(2), close(2),  
ioctl(2)



## 9.4. ngclcuIoctl(3)

### NAME

ngclcuIoctl - Send a control command to a NGCLCU I/O-device

### SYNOPSIS

```
#include "ngclcu.h"
```

```
int ngclcuIoctl(int channel, int command, void *argument);
```

### DESCRIPTION

This routine is called when a control command is send to an NGC LCU I/O device via ioctl(). It validates the command request, performs a semaphore protection if required and calls the command procedure to be executed.

channel - channel opened to the NGCLCU I/O-device  
command - number identifying the operation to be performed by the driver  
argument - address of the command argument or NULL if no argument is used. For commands needing more multiple arguments it is the address of a data structure which contains those.

Valid commands:

ngclcuCMD\_FREE\_DEVICE - free a device;  
The argument is ignored;

ngclcuCMD\_RESET\_DEVICE - reset a device;  
The argument is ignored;

ngclcuCMD\_READ\_REG - read from register;  
The argument is a pointer to an ngclcuREG object:

```
    UINT32 reg;          - register (offset)  
    UINT32 value;       - value
```

ngclcuCMD\_WRITE\_REG - write to register;  
The argument is a pointer to an ngclcuREG object:

```
    UINT32 reg;          - register (offset)  
    UINT32 value;       - value
```

ngclcuCMD\_READ\_PCI - read from PCI configuration register;  
The argument is a pointer to an ngclcuREG object:

```
    UINT32 reg;          - register (offset)  
    UINT32 value;       - value
```

ngclcuCMD\_WRITE\_PCI - write to PCI configuration register;  
The argument is a pointer to an ngclcuREG object:



```
UINT32 reg;      - register (offset)
UINT32 value;    - value
```

ngclcuCMD\_ACK\_INT - acknowledge that device interrupt is handled;  
The argument is ignored;

ngclcuCMD\_ATTACH\_INT - attach to device data interrupt;  
The argument is a pointer to an ngclcuCFG\_INT object:

```
SEM_ID      semISR; - interrupt service semaphore
ngclcuUSR_ISR *userISR; - user routine at interrupt level
void        *userArg; - pointer to user argument
BOOL        trigger; - trigger task on interrupt
```

If <userISR> is not a NULL pointer the user routine is called at interrupt level. The user routine is called with three arguments: a pointer to the received data buffer, the size of the buffer and the given <userArg>:

```
void userIsr(void *buffer, int dsize, void *userArg);
```

If <trigger> is set to TRUE the returned semaphore semISR can be used to trigger a routine at task level. If <userISR> is a NULL pointer semISR is always given in the interrupt service routine regardless of the value of <trigger>.

ngclcuCMD\_DETACH\_INT - detach from device data interrupt;  
The argument is ignored;

ngclcuCMD\_INT\_ENABLE - enable interrupt generation on NGCLCU-board;  
The argument is ignored;

ngclcuCMD\_INT\_DISABLE - disable interrupt generation on NGCLCU-board;  
The argument is ignored;

ngclcuCMD\_STATUS - get transfer status;  
The argument returns a pointer to an ngclcuSTATUS object:

```
int pIdx;      - buffer index for process
int tstat;     - transfer status - this is one of:

    ngclcuTSTAT_OK      - successful completion
    ngclcuTSTAT_OVERFLOW - FIFO overflow
    ngclcuTSTAT_ERROR   - link i/o-error

BOOL overrun;  - buffer overrun flag
BOOL attached; - attached to interrupt
```

ngclcuCMD\_HWSTATUS - get hardware status;  
The argument returns a pointer to an ngclcuHWSTATUS object:





```
UINT32 hwStatus; - hardware status
UINT32 revision; - revision
int dmaSize;     - DMA size
```

ngclcuCMD\_CONFIG - configure transfer;  
The argument is a pointer to an ngclcuCONFIG object:

```
void *data[2]; - pointers to user memory
int d_size;    - user buffer size in bytes
```

ngclcuCMD\_SIMTIME - set simulation interval;  
The argument is a pointer to an ngclcuSIMTIME object:

```
int simTime;    - simulation interval in microseconds
```

#### RETURN VALUES

```
lcudrvOK - successful completion
lcudrvERROR_INVALID_ARGUMENT - invalid channel number
lcudrvERROR_CHANNEL_NOT_OPEN - channel was not open
lcudrvERROR_INVALID_COMMAND - command code invalid
lcudrvERROR_ACCESS_CONFLICT - insufficient access rights
lcudrvERROR_TIMEOUT - access protection timed out
ngclcuERROR_SEMAPHORE_IVLD - semaphore not configured on device
ngclcuERROR_ATTACHED - device is already attached
ngclcuERROR_DETACHED - device is not yet attached
ngclcuERROR_NO_CFG - DMA not yet configured
ngclcuERROR_CFG_IVLD_DSIZE - invalid user buffer size
ngclcuERROR_CFG_IVLD_PTR - invalid user buffer pointer
```

#### SEE ALSO

ngclcuDrv(3), ngclcuDevCreate(3), ngclcuTools(3), ngclcuDevice(3),  
lcudrvOpen(3), open(2), close(2), ioctl(2)



## 9.5. ngclcuInterrupt(3)

### NAME

ngclcuInterrupt - Routines for the interrupt handling for the  
NGCLCU driver

### SYNOPSIS

```
#include "ngclcu.h"  
#include "ngclcuPrivate.h"
```

```
void ngclcuIsrData(int arg);
```

```
int ngclcuInitInterrupt(ngclcuDEVICE_DESCRIPTOR *devPtr);
```

### DESCRIPTION

These functions perform the interrupt handling of the NGCLCU driver.

ngclcuIsrData() is the interrupt service routine handling the device data interrupt. The routine takes care of the double buffer index and checks for overrun. Depending on the configuration the routine either triggers a task via semaphore or calls a user-defined interrupt service routine (or does both). The user-defined interrupt service routine uses the syntax:

```
void userIsr(void *buffer, int dsize, void *userArg);
```

If applicable the function is passed to the driver via the ioctl system call.

ngclcuInitInterrupt() initializes the interrupt data structures and connects the ngclcuIsrData() data interrupt service routine. Afterwards the data interrupt is enabled.

### RETURN VALUES

Unless specified otherwise:

OK	- successful completion
ERROR	- general error
lcudrvERROR_NO_SEMAPHORE	- semaphore creation failed
ngclcuERROR_INT_NO_HANDLER	- no interrupt handler

### SEE ALSO

ngclcuOpen(3), ngclcuClose(3), ngclcuIoctl(3), ngclcuDevCreate(3),  
ngclcuDrv(3)



## 9.6. ngclcuDevice(3)

### NAME

ngclcuDevice - Device layer routines for the NGCLCU driver

### SYNOPSIS

```
#include "ngclcu.h"
#include "ngclcuPrivate.h"

UINT32 ngclcu2Dev(UINT32 x);

UINT32 ngclcuPciAddr(void *buffer);

int ngclcuPciInit(ngclcuDEVICE_DESCRIPTOR *devPtr, void *baseAddress);

int ngclcuDmaReset(ngclcuDEVICE_DESCRIPTOR *devPtr);

void ngclcuDmaDescrSetup(ngclcu_desc_t *desc, void *dmaAddr, int dmaSize,
                        ngclcu_desc_t *next, int intr);

int ngclcuDmaNumDescr(int size, blockSize);

int ngclcuDmaCfg(ngclcuDEVICE_DESCRIPTOR *devPtr);

int ngclcuDmaStart(ngclcuDEVICE_DESCRIPTOR *devPtr);

int ngclcuDmaAbort(ngclcuDEVICE_DESCRIPTOR *devPtr);

int ngclcuDmaEnableInterrupt(ngclcuDEVICE_DESCRIPTOR *devPtr);

int ngclcuDmaDisableInterrupt(ngclcuDEVICE_DESCRIPTOR *devPtr);
```

### DESCRIPTION

ngclcu2Dev() is an inline function to convert a 32-bit CPU word into a register value (and vice versa).

ngclcuPciAddr() returns the PCI-bus address for <buffer>.

ngclcuPciInit() initializes the PCI-bus device.

ngclcuDmaReset() clears all DMA fifos.

ngclcuDmaDescrSetup() sets up a complete DMA-descriptor block. The parameter <next> contains the address of the next descriptor block. If <intr> is not zero then an interrupt is generated after the DMA associated to this descriptor has completed.

ngclcuDmaNumDescr() returns the number of DMA descriptor blocks needed for a DMA of the given <size>. The <blockSize> defines the maximum size of one DMA scatter-gather buffer.

ngclcuDmaCfg() configures the DMA descriptors for the sustained scatter/gather DMA.

ngclcuDmaStart() starts the sustained DMA.



`ngclcuDmaAbort()` aborts the sustained DMA.

`ngclcuDmaEnableInterrupt()` enables the interrupts from the devices.

`ngclcuDmaDisableInterrupt()` disables the interrupts from the devices.

#### RETURN VALUES

Unless specified otherwise:

<code>OK</code>	- successful completion
<code>ERROR</code>	- general error
<code>lcudrvERROR_TIMEOUT</code>	- access protection timed out
<code>lcudrvERROR_NO_MEMORY</code>	- memory allocation failed
<code>ngclcuERROR_CFG_IVLD_DSIZE</code>	- invalid buffer size
<code>ngclcuERROR_PCI_NOT_FOUND</code>	- device not found on PCI-Bus
<code>ngclcuERROR_PCI_MEM_MAP</code>	- PCI memory mapping failed

#### SEE ALSO

`ngclcuOpen(3)`, `ngclcuClose(3)`, `ngclcuIoctl(3)`, `ngclcuDrv(3)`,  
`ngclcuDevCreate(3)`, `ngclcuTools(3)`, `open(2)`, `close(2)`, `ioctl(2)`



## 9.7. ngclcuTools(3)

### NAME

ngclcuTools - Tools for the NGCLCU driver

### SYNOPSIS

```
#include "ngclcu.h"

const char *ngclcuTransferStat2Str(u_long transferStatus);

const char *ngclcuErrorGet(int status);

void *ngclcuMalloc(unsigned int size);

void ngclcuFree(void *buffer);

void ngclcuSwap2(void *buffer, int length);

void ngclcuSwap4(void *buffer, int length);

void ngclcuVersion(void);

void ngclcuDevShow(void);

STATUS ngclcuStatus(int instance);

STATUS ngclcuClr(int instance);

STATUS ngclcuReadReg(UINT32 reg, int instance);

STATUS ngclcuWriteReg(UINT32 reg, UINT32 value, int instance);

STATUS ngclcuIntOff(int instance);

STATUS ngclcuIntOn(int instance);
```

### DESCRIPTION

ngclcuTransferStat2Str() returns a pointer to a string translating the transfer status passed by <tstat>.

ngclcuErrorGet() returns a pointer to a string translating the <status> (result code of NGCLCU I/O function calls).

ngclcuMalloc() allocates <size> bytes of properly aligned memory and returns a pointer to it.

ngclcuFree() frees a buffer that has previously been allocated with the ngclcuMalloc() function.

ngclcuSwap2() changes the byte order of the 16-bit numbers in <buffer> from 1,2 to 2,1 (Big endian to little endian). <length> is the number of 16-bit items of <buffer>.

ngclcuSwap4() changes the byte order of the 32-bit numbers in <buffer> from 1,2,3,4 to 4,3,2,1 (Big endian to little endian). <length> is the number of 32-bit items (integer, single prec. float) of <buffer>.



`ngclcuVersion()` prints the driver version.

`ngclcuDevShow()` prints a list of NGCLCU I/O-devices controlled by the driver.

`ngclcuStatus()` prints the current device status on stdout. The device is specified by its instance number.

`ngclcuClr()` resets the FIFOs on a device specified by its instance.

`ngclcuReadReg()` reads a register value from a device specified by its instance.

`ngclcuWriteReg()` writes a value to a register on a device specified by its instance.

`ngclcuIntOff()` disables the interrupts from a device specified by its instance.

`ngclcuIntOn()` enables the interrupts from a device specified by its instance.

#### RETURN VALUES

Unless specified otherwise:

- OK - successful completion
- ERROR - general error



## 9.8. ngclcuCapture(3)

### NAME

ngclcuCapture - Data capture functions for the NGCLCU driver

### SYNOPSIS

```
#include "ngclcuCapture.h"
```

```
int ngclcuInit(const char *devName, char *erms);
```

```
int ngclcuReset(int channel, char *erms);
```

```
int ngclcuConfig(int channel, void **buffer, int size, int blockSize,  
char *erms);
```

```
SEM_ID ngclcuStart(int channel, ngclcuUSR_ISR *userISR,  
void *userArg, char *erms);
```

```
int ngclcuStop(int channel, char *erms);
```

```
int ngclcuWait(int channel, SEM_ID semISR, int timeout, char *erms);
```

```
int ngclcuAck(int channel, char *erms);
```

```
int ngclcuRelease(int channel, char *erms);
```

### DESCRIPTION

ngclcuInit() returns a channel to an NGCLCU device specified by <devName>. If the channel could not be opened (-1) is returned.

ngclcuReset() resets the state-machine of the NGCLCU device. FIFO and FIFO-status are cleared, but all control configuration is kept.

ngclcuConfig() configures a data capture via an NGCLCU device specified by a channel returned by ngclcuInit(). The (double-)buffer has to be allocated by the calling routine. If a NULL pointer is passed as buffer, the previous buffers are used. If no buffers have been configured with a prior call an error is returned. The <size> of the buffer is given in bytes. The <blockSize> defines the maximum size of one DMA scatter-gather buffer. When set to zero, a default DMA block-size is used.

ngclcuStart() starts the data capture loop and returns a semaphore to be passed to following ngclcuWait() calls. If <userISR> is not NULL the user defined routine is called at interrupt level with three arguments: a pointer to the received data buffer, the size of the buffer and the given <userArg>:

```
void userIsr(void *buffer, int dsize, void *userArg);
```

ngclcuStop() aborts the data capture loop (asynchronous call).

ngclcuWait() waits with timeout (in ticks) for the next data buffer and returns the buffer index that has to be used for the processing. If <timeout> is WAIT\_FOREVER the routine waits until the transfer



has completed or an error/abort occurred. The function returns a negative value (error code) if an error occurred. If the transfer has been aborted `ngclcuERROR_ABORTED` is returned as error code (this should be caught by the calling application).

`ngclcuAck()` has to be called when all processing is done with the current buffer.

`ngclcuRelease()` should be called to release (and close) a channel to an NGCLCU device.

#### RETURN VALUES

Unless specified otherwise:

OK - successful completion  
ERROR - general error

#### SEE ALSO

`ngclcuIoctl(3)`, `ngclcuDrv(3)`, `ngclcuDevCreate(3)`,  
`open(2)`, `close(2)`, `ioctl(2)`





## 9.9. ngclcuTask(3)

### NAME

ngclcuTask - Task control for the NGCLCU driver

### SYNOPSIS

```
#include "ngclcuTask.h"
```

```
int ngclcuTaskInit(void);
```

```
int ngclcuTaskRegister(const char *name,  
                       FUNCPTR entryPt,  
                       int options,  
                       int stackSize,  
                       int numParam,  
                       ngclcuPARAM *param,  
                       int protectionTimeout,  
                       char *erms);
```

```
void ngclcuTaskFree(int id);
```

```
void ngclcuTaskFreeAll(void);
```

```
int ngclcuSp(FUNCPTR entryPt);
```

```
int ngclcuTaskId(const char *name);
```

```
BOOL ngclcuTaskStart(int id, int channel);
```

```
BOOL ngclcuTaskCont(int id, BOOL brk);
```

```
void *ngclcuTaskParam(int id);
```

```
void ngclcuTaskError(int id, const char *erms);
```

```
void ngclcuTaskReqOut(ngclcuFRAME frame, int id);
```

```
int ngclcuTaskArgSet(const char *arg);
```

```
char **ngclcuTaskArgGet(void);
```

```
void ngclcuTaskFlags(int id, BOOL *resetFlag, BOOL *endFlag);
```

```
void *ngclcuTaskMem(int memId, int size, BOOL *valid);
```

```
int ngclcuTaskMemValidate(int memId, BOOL valid);
```

```
int ngclcuTaskMemSize(int memId);
```

### DESCRIPTION

ngclcuTaskInit() initializes the task handling and should be called once when installing the module. Multiple calls to ngclcuTaskInit without deleting the task-handling with ngclcuTaskFreeAll() have no effect.

ngclcuTaskRegister() registers a task for execution via the



ngclcuServer. Parameters are the same as for taskSpawn. The priority is set by the server. Additionally a parameter array containing <numParam> parameter structures has to be passed:

```
char name[64]; - parameter name
int  value;    - parameter (default) value
int  type;     - parameter type
```

Supported parameter types are ngclcuPTYPE\_INT (32 bit integer) and ngclcuPTYPE\_FLOAT (32 bit floating point). The protection timeout for the parameters is specified in ticks. An internal task-id is returned which is also passed to the task as (only) parameter. This id has to be used for all other calls to the ngclcu task interface functions. The task is spawned, when the server receives a start command.

ngclcuTaskFree() removes a task from the registry.

ngclcuTaskFreeAll() removes all tasks from the registry and deletes the task-handling.

ngclcuSp() spawns the specified NGCLCU-application task with default parameters.

ngclcuTaskId() returns the internal id of the task registered with <name>.

ngclcuTaskStart() synchronizes to the start command. If the function returns FALSE, then the task should terminate.

ngclcuTaskCont() synchronizes to the stop command. By setting <brk> to FALSE the task signals to the server, that it will not exit but wait again for the next start command. If <brk> is TRUE the task should terminate immediately. If the function returns FALSE, the task should also terminate.

ngclcuTaskParam() returns a pointer to a task specific parameter structure. The pointer should be casted to a structure with integer/floating-point values in the same order as given in the <param> argument to ngclcuTaskRegister(). This function can be called at any time and returns the actual parameter setup values.

ngclcuTaskError() puts an error message to the error-stack. This should be done only for fatal-errors, which require that the task has to be stopped/restarted.

ngclcuReqOut() passes a data frame to the data transfer task(s). The function returns immediately. The transfer of <frame> is done asynchronously if the function returns TRUE. If FALSE is returned then no action has been taken.

The frame structure (ngclcuFRAME) is defined as follows:

```
ngclcuFRAME_H  h;          - header structure
char           *fbuf;      - frame buffer
```

The frame header structure (ngclcuFRAME\_H) is defined as follows:

```
int dtype;      - data type
```



```
int ftype;      - frame type
int start_x;    - window start-x
int start_y;    - window start-y
int nx;         - window nx
int ny;         - window ny
int scal;       - scale factor
int cnt;        - frame counter
int setupId;    - setup-id
int err;        - error-id
int overrun;    - overrun flag
int frames;     - available frame types
int tx;         - track point x
int ty;         - track point y
```

ngclcuTaskArgSet() sets a global argument string. The arguments are separated by white spaces. The current task can retrieve a pointer to the argument list by calling ngclcuTaskArgGet().

ngclcuTaskArgGet() returns the NULL terminated argument list.

ngclcuTaskFlags() returns the run-control flags of the task specified by <id>. The flags are set to FALSE after each call.

ngclcuTaskMem() returns a pointer to data-set memory identified by <memId>. The <memId> has to be a single-bit value. The id-assignment is the responsibility of the task application(s). If <size> is greater than zero, <size> bytes of memory are allocated. If <size> is equal to ngclcuTASK\_MEM\_FREE, the data-set is removed. In this case a NULL pointer is returned. If <memId> is equal to ngclcuTASK\_MEM\_ALL, all data-sets are removed (in this case <valid> can be specified as NULL pointer). If the function is called again with the same <memId> but different size, then the memory of the data-set is reallocated and marked as invalid. If the memory could not be allocated a NULL pointer is returned. If <size> is equal to ngclcuTASK\_MEM\_NO\_CHANGE only the buffer belonging to the specified <memId> (or a NULL pointer if not yet initialized) is returned. If the memory of the data-set has been validated once by an explicit call to ngclcuTaskMemValidate() the <valid> flag is set to TRUE. A task application must never use the memory of a data-set marked as invalid.

ngclcuTaskMemValidate() validates/invalidates the memory of all data-sets given in <memIdMask> depending on the value of <valid>. <memIdMask> is a mask with all single-bit memory id's set, for which the operation has to be performed.

ngclcuTaskMemSize() returns the size (in bytes) of the memory allocated for a specific data-set or -1 if the data-set has not yet been configured.

#### RETURN VALUES

Unless specified otherwise:

```
OK      - successful completion
ERROR   - general error
```

#### SEE ALSO

ngclcuCapture(3), ngclcuServer(1), taskSpawn(2), sp(2)



## 9.10. ngclcuServer(1)

### NAME

ngclcuServer - Data capture control server for the NGCLCU interface

### SYNOPSIS

```
#include "ngclcuTask.h"
```

```
STATUS ngclcuServer(int cmdPort, int dataPort);
```

```
int ngclcuSrvCmdPort(void);
```

```
int ngclcuSrvDataPort(void);
```

```
int ngclcuSrvErrorPort(void);
```

```
void ngclcuSrvAbort(void);
```

### DESCRIPTION

ngclcuServer() is the data capture control server for the NGCLCU interface. If <cmdPort> is zero a free port number is chosen by the operating system, which can be retrieved with the ngclcuSrvCmdPort() function. If <dataPort> is zero and cmdPort is not zero, then the data port number is computed as command port number plus one. If both port numbers are zero, a free port number is chosen also for the data port, which can be retrieved with the ngclcuSrvDataPort() function.

ngclcuSrvCmdPort() returns the actual command server port number.

ngclcuSrvDataPort() returns the actual data server port number.

ngclcuSrvErrPort() returns the actual error stack server port number.

ngclcuSrvAbort() aborts the data capture control server.

### RETURN VALUES

Unless specified otherwise:

```
OK          - successful completion  
ERROR      - general error
```

### SEE ALSO

ngclcuDrv(3), ngclcuDevCreate(3), ngclcuCapture(3), ngclcuTask(3),  
ngclcuSock(3)