Smart Cameras made in Germany

Vision Components

The Smart Camera People

VCRT 5.0 Software Manual

Operation System Functions

Revision 5.30.7 Apr 20 2012 MS
Document name: VCRT5.pdf

Vision Components GmbH
Ottostr 2
76275 Ettlingen
Telefon +49 (0)7243 2167-0
Fax +49 (0)7243 2167-11

Foreword and Disclaimer

This documentation has been prepared with most possible care. However, Vision Components GmbH does not take any liability for possible errors. In the interest of progress, Vision Components GmbH reserves the right to perform technical changes without further notice.
Please notify support@vision-components.com if you become aware of any errors in this manual or if a certain topic requires more detailed documentation.
This manual is intended for information of Vision Component's customers only. Any publication of this document or parts thereof requires written permission by Vision Components GmbH.

Please also consult the following resources for further reference:

Description	**Titel on** www.vision-comp.com	**Download from Area**
Getting Started VC Smart Cameras	Getting Started VC Smart Cameras with TI DSP	Support & Service > Download Center
Einführungshandbuch VC Smart Kameras	Schnellstart VC Smart Kameras mit TI DSP	Support & Service > Download Center
Introduction à l'utilisation des caméras Vision Components	Démarrage rapide Smart Cameras Vision Components	Support & Service > Download Center
Introduction to VC Smart Camera Programming	Programming Tutorial Basics	Support & Service > Download Center
Demo programs used in Programming Tutorial Basics	Tutorial_Code	Support & Service > Download Center
VC4XXX Hardware Manual	VC40XX Smart Cameras Hardware Documentation	Support & Service > Download Center
VCSBC4XXX Single Board Smart Camera Hardware Manual	VCSBC4018 and VCSBC4016 Manual	Support & Service > Download Center
VCRT Operation System TCP/IP Functions Manual	VCRT 5.0 TCP/IP Manual	Support & Service > Download Center
VCLIB 2.0 /3.0 Image Processing Library Manual	VCLIB 2.0/ 3.0 Software Manual	Support & Service > Download Center

Note:

→ **This document is valid for VC Smart Cameras with Texas Instrument DSP only!**

→ **The TCP/ IP Function are now described in a separate document (see refererences).**

The Light bulb highlights hints and ideas that may be helpful for a development.

This warning sign alerts of possible pitfalls to avoid. Please pay careful attention to sections marked with this sign.

Table of Contents

1 General Information

The VC Series cameras are compact, light-weight black-and-white or color video cameras with video memory and a frame processor. They integrate a high-resolution CCD/CMOS sensor with a fast frame-processing signal processor. A dynamic RAM is used to store data and video frames. Interfaces allow communication with the outside world. The cameras set standards for performance and integration density.

These cameras are built for industrial applications. High goals were set as regards the frame resolution, the sturdiness of the casing, and the electromagnetic compatibility, as mere examples. The cameras are insensitive to vibrations and shocks, while permitting precise measurements and tests. They are ideally suited as OEM cameras for mechanical engineering applications.

This documentation describes the cameras' **software**, especially the operating system functions and general functions. However, in many cases the **hardware** documentation is decisive. Special function libraries are also documented separately. Please consult the corresponding manuals.

For the following topics refer to the “VC20XX VC4XXX VC6XXX VC7XXX Installation Manual”:

- Overview of Vision Components Development Software/ Licencing/ SW Registration and Updates
- Setup and use of Code Composer Studio
- SW Compilation using CCS
- Location of Header, Libs, Utilities and Demo Files on your PC after Installation of the VC SDK-TI
- Cabling Overview
- Communication with the VC Smart Camera, Uploading of Programs
- Overview of the Camera Shell (for detailed information refer to this manual)
- Structure of the Vision Components Web Site including the Support section
- Trouble Shooting Guide camera / PC comunication

For a Programming Tutorial including detailed descriptions of sample code refer to the Prog_Tut.pdf (Programming Tutorial VC20XX,VC4XXX,VC6XXX and VC7XXX Smart Cameras).

Please also refer to section 17 for a list of previously undocumented VCRT Functions.

2 Tasks of the Operating System

The operating system VC/RT controls all of the camera's elementary functions. It also provides the user with a command interpreter (the "shell") for easy user access to all resources. It supports the user in the debugging and test phase. VC/RT is a real-time multitasking operating system, i.e. it can execute several tasks in parallel and it can guarantee execution times for time-critical tasks. VC/RT contains a fully-featured TCP/IP stack which allows communication using a variety of modern communication standards like TELNET, FTP or HTTP.

The following table compares the properties of VC/RT to those of other operating systems

Property	VC/RT	MS-DOS	OS/9	UNIX/LINUX	WINDOWS
Real-time capable	yes	no	yes	no	no
Multitasking	yes	no	yes	yes	yes
Timeslice	1 msec	---	10 msec	1 – 10 msec	50 msec
Filesystem tolerant to power interruption	yes	no	yes	no	no
Royalties	one-time*)	per installation	per installation	LINUX: none	per installation

*) one-time license per developer workstation, no royalties
The interface to the VC/RT system and file utilities is compatible to POSIX and to a high degree to UNIX.

3 VC/RT Resources

The main task of an operating system is to administer the processor's resources. However, an operating system for a video camera must control somewhat "uncommon" resources:

Resource	Functions
CCD/CMOS sensor	Picture taking and reproduction, various control functions
Frame output	Control of the display and overlay outputs
Flash EPROM	Loading of VC/RT kernel / File access
SD card / Multi-media card	File access
DRAM	Accessing and managing memory, allocating and releasing memory
RS232 interface	Data buffering and background I/O operations
Ethernet	Fully featured Highspeed TCP/IP stack / socket communication
Interrupts	Control of the various interrupt sources

There are library programs for most of the above operating system functions, which interface to the user program (C program).

VC/RT consists of the following components:

- The kernel
- The shell
- TELNET server
- FTP server
- HTTP server
- Various routines which can be linked to the user program.

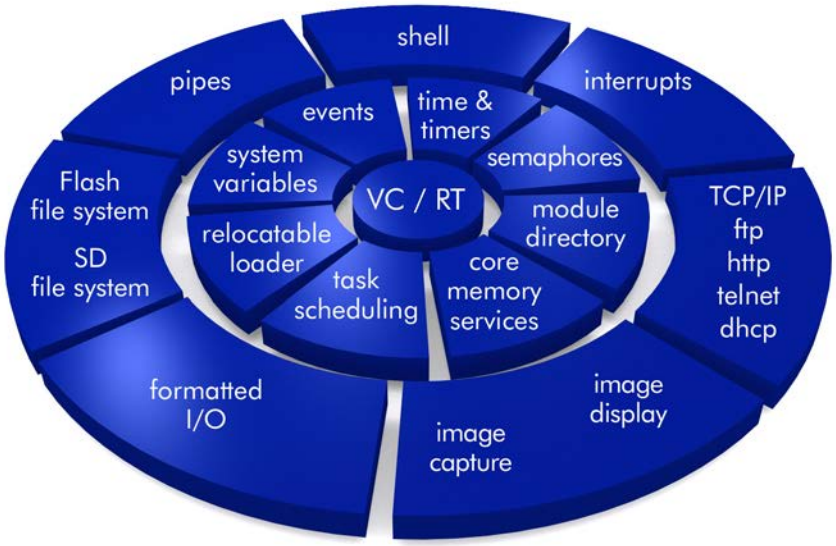


Fig. 1: VC/RT functionality

4 The VC/RT Kernel

The kernel is located permanently at addresses in dynamic RAM (DRAM)

```
VC20XX and VC4XXX : 0xA0000000  16MB (VC20XX) SD-RAM
                                32MB (VC40XX) SD-RAM
                                64MB (VC44XX) SD-RAM
VC6XXX and VC7XXX : 0x80000000 128MB (VC6XXX) DDR2-RAM
VC64XX models     : 0xE0000000 256MB (VC64XX) DDR2-RAM
```

It thus occupies 1 MBytes of memory. (The memory model is described in Organization of the DRAM)

The kernel consists of the following components:

- During power-up or reset, the loader loads the shell (filename: "shell").
- Interrupt-controlled routines for time management. Via an interrupt, all time-related functions are controlled once per millisecond.
- Interrupt-controlled routines for all communication channels (serial or Ethernet).
- Interrupt-controlled routines for the PLC inputs/outputs. On any change of the camera's inputs an interrupt is generated with which the status of the input lines is copied to the PLCIN system variable. Other interrupts detect power failure conditions
- DMA-controlled routines for taking and displaying pictures. Via DMA, all frame-related display and capture functions are controlled.
- DMA-controlled routines and file-system for SD card / multi-media card access.
- DMA-controlled routines for ETHERNET communication
- Integrated TCP/IP stack with TELNET, FTP and HTTP servers.
- System variables allow access and modification of operating modes
- With the VCRT event system, user programs can wait for events like image capture without wasting CPU time
- The relocation loader allows user programs to be loaded in memory at variable locations depending on availability of memory space.

5 The Shell ("shell")

The shell is a program loaded by the loader task. The shell communicates with the user via the serial interface. (A PC with a communications program, such as TERATERM, is commonly used for this.)
As is common with most operating systems, commands can be entered (with or without parameters) and are interpreted by the shell.

The shell itself contains a number of useful commands which can be executed directly. A built-in help command (called by entering **he**) provides a quick overview of these functions.

The shell also determines if entered commands must be executed from the flash EPROM or SD/Multimedia-card (the command could also be a user program or batch file, for instance).
In this case, the program is loaded, the command string is transferred and the program is started. The shell is reloaded to main memory after the program terminates.

In addition to being the user interface, which allows entering commands, loading and executing programs, the shell provides the following features:

1. execution of batch files
any shell command or any available program name may be placed in an ASCII-file which may be executed simply by typing it's name.

example:

batch file commands	comment (not part of the batch file)
`bd 19200`	`set baudrate to 19200 bauds`
`#st`	`execute self-test function (sector 0 program)`
`userpg1`	`execute user program userpg1`
`jl img`	`display JPEG image img`
`autoexec`	`execute batch file autoexec`

Note: do not call batch files recursively

2. any shell command may be invoked by a running program simply as parameter for the program “shell” (in-line mode)

example:

```
#include <vcrt.h>

/* argc=2 is the number of arguments in the
 * command line argv
 */
void main(int d, int argc, char *argv)
{
  ....
  exec("shell",2,"bd 19200");/* 2 parameters = bd + 19200  */
  ....
}
```

remark: calling a batch file with exec is also possible

example:

```
#include <vcrt.h>

void main(int d, int argc, char *argv)
{
...
exec("shell",1,"batch");      /* 1 parameter = batch        */
...
}
```

3. The shell itself may be called by a user program (e.g. to check memory usage, change shutter settings, etc.). You may resume operation of the calling program simply by typing 'ex'.

example:

```
#include <vcrt.h>

void main(int d, int argc, char *argv)
{
argc=0;                              /* shell is called without    */
*argv='\0';                          /* parameter                  */

exec("shell",argc,argv);
}
```

Note, that the command line buffer argv of the previous shell is used. This saves valuable memory space. Otherwise a command line buffer with 80 elements char argc[80] must be supplied on the stack or heap.

5.1 Description of the Shell Commands

The shell contains the following internal commands (in alphabetical order):
(bold writing indicates changes or new commands resp. older VCRT versions)

bd	set baud rate	bd <baudrate>
cd	change data directory	cd <path>
cx	change execution directory	cx <path>
copy	copy a file	copy <source path> [<dest path>]
del	delete file	del <path>
dir	directory of Files	dir [<option>] [<path>]
disp	**switch display modes**	**disp [<option>] [<mode>]**
dd	DMEM Display	dd <addr>||<range>
dwn	download file to PC	dwn <path>
er	erase complete flash eprom	er
ex	exit from shell	ex
fmt	**format media card**	**fmt [<size> [<clustersize>]]**
?	help	? [<name>]
he	help	he [<name>]
help	help	help [<name>]
ht	hardware test	ht
js	jpeg store	js <path>
jl	jpeg load	jl <path>
kill	**delete task**	**kill <PID>**
kl	**kernel log**	**kl**
lo	load S records	lo
mdir	**display module directory**	**mdir [<option>]**
mem	**display memory usage**	**mem [<option>] [<PID>]**
mkdir	make directory	mkdir <path>
ping	**test IP connection**	**ping <IP-address>**
pk	pack flash memory	pk
procs	print task list	procs
sh	set shutter value	sh <number>
time	time and date command	time [<option>]
tp	take picture	tp
type	type ASCII file	type <path>
ver	print software version	ver
vd	video modes	vd [[<option>] <frame number>]
wb	**whitebalance**	**wb**

bd **set baud rate for the serial interface**

synopsis `bd <baudrate>`

description The baud rate for the serial interface can be changed with bd.The parameter is a decimal specifying the baudrate. Non-standard values are also supported. The maximum baudrate is 115200, the minimum value is 300. Settings that cannot be changed are parity (always: NONE), stop bit (always: 1) and data bits (always: 8).

example: `bd 19200`

cd **change path for working directory**

synopsis `cd <path>`

description This command changes the path of the working directory. A valid path consists of a drivename (fd: or md:) and an optional subdirectory structure.

examples

`cd md:/my_directory/`	selects directory "my_directory" on multi-media card
`cd fd:`	selects flash-EPROM
`cd fd:/user/`	selects flash-EPROM (user sectors)
`cd fd:/sys/`	selects flash-EPROM (system sectors)

cx **change path for execution directory**

synopsis `cx <path>`

description This command changes the path of the execution directory. A valid path consists of a drivename (fd: or md:) and an optional subdirectory structure.

examples

`cx md:/my_directory/`	selects directory "my_directory" on multi-media card
`cx fd:`	selects flash-EPROM (user sectors)
`cx fd:/user/`	selects flash-EPROM (user sectors)
`cx fd:/sys/`	selects flash-EPROM (system sctrs.)

copy **copy file**

synopsis `copy <sourcepath> [<destpath>]`

description This command copies a file to a different location. A valid path consists of a drivename (fd: or md:), a subdirectory structure and a file-name. If the destination path is ommited, the current directory is assumed.

examples

`copy md:/my_directory/test.jpg`	copies test.jpg from directory "my_directory" on MMC/SD-Card to current data directory
`copy fd:test.jpg md:/test.jpg`	copies file test.jpg from flash to MMC/SD-Card

del **delete file**

synopsis

```
del <path>
```

description

A file can be deleted with the command del. A valid path consists of a drivename (fd: or md:), a subdirectory structure and a file-name.
For the Flash EPROM (fd:), the file itself stays in the flash EPROM. It is only marked as "deleted".

A "deleted" file still takes up space in flash memory. This memory space can be used for other purposes after reorganizing the complete file system with the 'pk' (pack) command or after erasing all files with the command er.

dir **display directory of files**

synopsis

```
dir [<option>][<path>]
```

description

The command dir creates a list of all files in the directory. The directory path may either be specified directly or indirectly using options. A valid path consists of a drivename (fd: or md:) and the subdirectory structure.

The following information is shown:

1. file name and extension
2. total length in bytes (decimal)
3. time and date of last write access(not shown for fd:)

Calling dir without options lists all files in the default directory chosen with cd

Options:

-x	list system files (in sector 0) on fd:
-a	list all files including deleted files on fd:

examples

`dir`	Outputs a list of files of the working directory
`dir -x`	Flash system directory
`dir md:`	Directory of device md:
`dir md:/sub`	List subdirectory md:/sub

disp **switch display modes / gamma / period**

synopsis `disp [<option>][<mode>]`

description The command `disp` changes the display mode and display period It also allows to show and set the gamma value for all cameras. There are several options, some of which are not available for black-and-white cameras:

-c	change color mode	(color cameras only)
-g	change gamma correction	
-p	change display period	
-a	display active (1) / inactive (0)	

option –c:

This option changes the color mode for the display. Images can be displayed in a variety of color formats including grey value output (black-and-white) and YUV format (YCbCr)

```
0     IDLE
1     GREY
2     RGB
3     BAYER
4     BAYERGREY
5     YCBCR
```

example `disp -c 5` change to YCbCr display

option –g:

This option allows to set the gamma correction for the display. Display monitors normally have a non-linear, mostly logarithmic transfer function. You can enter 100 times gamma with this command.
The default is 0.6 (set value is gamma*100 = 60). Called without a parameter, the current value is shown.

example `disp -g 100` change gamma to 1 (default is 0.6)

option –p:

This option changes the refresh rate (`DISP_PERIOD`) of the display. Display refresh adds a certain overhead, which slows down the processing power of the CPU. For black-and-white cameras, this overhead is mostly negligible, since only memory transfers are involved, the CPU running at full speed. For color cameras, however, the CPU must calculate the color conversion, which is quite time consuming. A color conversion may take up to 60 milliseconds depending on color mode and DSP type and speed grade. The refresh rate is defined in units of the vertical retrace time which is typically 14 milliseconds

for an SVGA display. This command also changes the system variable `DISP_PERIOD`.

The default for `DISP_PERIOD` is 20. Called without a parameter, the current value is shown.

option -a:

`disp –a 0` switches the display off. For VC20xx smart cameras this means that there is no update of the video refresh buffer, i.e. the last image or video graphic is “frozen”. For VC40xx and VC44xx smart cameras the video output is simply black. In both cases, a switched-off video display does not consume any memory bandwidth and therefore results in maximum computational performance.
`disp –a 1` switches the display an. This is the default state.
This option changes the refresh system variable `DISP_ACTIVE`.

Example `disp -p 10` change refresh period to 140 milliseconds

dwn **download file to PC / flash EPROM**

synopsis `dwn <path>`

description The command dwn sends a file in S-record format to a host PC. The command returns the following message:

```
please activate PC download function (e.g. PgDn-key)
press ESC to abort or any other key to continue
```

The user should then activate the download function of the terminal program. For PROCOMM this is done by pressing the PgDn key. Enter the protocol (ASCII) and file name.
Sending an arbitrary character (like RETURN) starts the sending procedure.

er **erase / format Flash EPROM**

synopsis `er`

description The entire flash EPROM can be physically erased (formatted) with the command `er` (except for the system sectors 0 - 15). It is first determined if the affected sector is already empty. If so, this is reported and the sector will not be erased.

It's not possible any more to erase indiviual sectors from the shell. For compatibility reasons, the function erase() is still available. Please use file based functions instead.

ex **exit from shell**

synopsis `ex`

description This command is used to return from a shell to the calling program. Simply type 'ex' and control will be passed to the calling program. If the shell has not been called by a user program, ex has no effect.

The former paths of "cd" and "cx" are restored.

fmt **format media card**

synopsis `fmt [<size in MB> [<clustersize in blocks>]]`

description This command is used to format the media device (the built-in multi media card or SD-card). The default size is 16MB, i.e. calling the command without a parameter will format the media card to 16MB regardless of its real size. For larger sizes, the command may be called with its size as a parameter: 16, 32, 64, 128, … 1024 are allowed values for the size. If the value does not match a value from the list, the default 16MB will be taken.
The second optional parameter is the clustersize in blocks (each block has 512 bytes). This value must be equal to or larger than 32.

he **help command**

synopsis `he [<name>], or: ?, help`

description **he** without parameters displays a list of all available commands. If the name of a command from the list is included as a parameter, **he** displays the syntax for the corresponding command.

ht **hardware test**

synopsis `ht`

description The function **ht** tests the hardware and displays a test screen. If an error occurs during the test, this will be reported.

`ht` performs the following individual tests:

1. processor test (mainly functionality of internal registers, memory, etc.)
2. DRAM test
3. ID and serial number
4. file system
5. VC/RT version of files (incompatible files will be deleted)
6. write a test pattern to image #0

Tests (1) through (5) are also executed on power-up as a self-test. If test (3) fails (e.g. due to manipulations of the serial number) the system will be halted. All other errors will be reported.

The test screen consists of the following test areas:

image data memory
gray wedge
4 alignment markers

overlay
- image boundary (yellow)
- cross hair (green)
- 4 centered frames of different size (blue, red, magenta)
- 1 circle for monitor adjustments (yellow)
- 4 translucent overlay areas (3 different colors = yellow, cyan, magenta)
- text: "Vision Components"

jl **jpeg load**

synopsis `jl <path>`

description Entering `jl <path>` will load a previously stored JPEG image file to the frame buffer.

example: `jl fd:/mylogo.jpg`

js **jpeg store**

synopsis `js <path>`

description Entering `js <path>` will store the complete image of the frame buffer (memory page 0) to the JPEG file <path> on the flash eprom. The quality factor for storing the image is 50%, which means that a data reduction of 10 to 20 may be assumed.

example: `js fd:/mylogo.jpg`

kill **delete task**

synopsis `kill <PID>`

description Entering `kill <PID>` will delete an active task with PID-number `PID` and remove it from the task list. Be sure not to delete vital system tasks with this command. You can get the task number using the `procs` command.

kl **kernel log**

synopsis `kl`

description This command outputs a “kernel log”, i.e. useful information that has been stored during the execution of the kernel. If you have questions concerning the kernel log output, please consult the Vision Components support.

lo **load S Records / flash EPROM**

synopsis `lo`

description Executable programs, ASCII files, binary data files, JPEG files, etc. can be loaded from the host computer (PC) to the flash EPROM with the command **lo**.
This command is especially important when developing programs. The program first finds the next free memory area in the flash EPROM, and the upload can begin. The data files must be sent (e.g. using the TERATERM communication program) in the S-Record HEX data format.

with this command, programs can only be stored on the FLASH Eprom, i.e. an upload to the media card / SD card is not possible.

You can also download programs efficiently using ftp on VC cameras with Ethernet. Refer to the “Getting Started VC Smart Cameras with TI DSP” Manual for details

mdir **display module directory usage**

synopsis `mdir [<option>][<MID>]`

description This command may be used to control the usage of the module directory. Entering mdir without option will display a summary of used modules.

Options: -v detailed display of modules in use

Entering mdir with a module ID (MID) as a parameter gives a detailed display of the module with the specified MID

examples

```
$mdir

display module directory
MID PID    STATE LINK SIZE       NAME
1   65545  2     0    0x1d58b    shell


$mdir 2
```

```
display module directory
MID PID    STATE LINK SIZE       NAME
2   65547  2     0    0x1d58b    shell

SECTION  ADDRESS      SIZE       ENTRY        STACKSIZE
0        0xa0494174   0x195a3    0xa04ac580   0x4000
3        0xa030a774   0x13b
5        0xa030b4b4   0xdbf
6        0xa030c294   0x208f
8        0xa030e334   0xc53
10       0xa030a8d4   0x143
11       0xa030aa34   0xe3
12       0xa030ab34   0x123
13       0xa030ac74   0xc3
```

mem **display memory usage**

synopsis `mem [<option>][<PID>]`

description This command may be used to show the memory usage of both the operating system and user programs e.g. for debugging purposes.

Entering mem without option will display a summary of used and free memory blocks.

options -v detailed display of memory segment usage

Entering `mem <PID>` lists the memory usage for the task with the process ID `PID` .

example

```
$mem 65546
display memory usage
ADDR         PID     SIZE         STATE   CHECKSUM
0xa0462280   65546   0x40         USED    OK
0xa0462400   65546   0x440        USED    OK

178 mem blocks (use -v to show all)
0x00167040 bytes in use ( 4%)
0x01b983a0 bytes free   (96%)
```

ping **test IP communication**

synopsis
```
ping <IP-address>
```

description The command ping tests the communication response of the IP device with `IP-address`. The command tests the communication in a loop until ESC is entered. `rtt` is the round-trip time, i.e. the time delay from sending the request to receiving the response.

example
```
$ping 192.168.0.99
ping host IP
192.168.0.99 seq=0 rtt=5 ms
192.168.0.99 seq=1 rtt=1 ms
192.168.0.99 seq=2 rtt=1 ms
192.168.0.99 seq=3 rtt=1 ms
192.168.0.99 seq=4 rtt=3 ms          <ESC>
$
```

pk **pack flash memory**

synopsis
```
pk
```

description The command pk physically purges deleted files from the flash eprom file system. The command allocates memory from DRAM, copies files to DRAM memory, while discarding deleted files, erases all previously used flash eprom sectors and then writes back the files to flash eprom.
Since the command may erase a large number of sectors, execution may take from 5 to 30 seconds, so **please be patient.**

The command will fail, if there is not enough memory available. This may happen if memory was allocated by a user program, but not freed.

procs **print task list**

synopsis
```
procs
```

description The command procs outputs a list of all tasks currently registered to the system. The command gives the following information for the task:

Task name	Process ID	state	priority	flags

The task state may be ACT = active or WAIT = waiting.
A higher value for priority, means that the task is ***lower*** in its priority.

time **display system time**

synopsis `time [<option>]`

description VC/RT for some cameras features a real time clock ("RTC") with battery backup. GMT (Greenwich Meantime) is stored internally, but any local time may be output by entering timezone and the daylight savings time flag.
Be sure to enter timezone and daylight saving time flag before changing the time setting.

The battery used is rechargeable. If fully loaded and temperatures are below 40 C it will keep the RTC working for at least 14 days . The RTC may function well for a much longer period depending on temperature, initial charge, battery age and device tolerances but this cannot be guaranteed. In the case of battery failure the time command will output:

```
low voltage detected
clock data may be invalid
```

In this case the RTC must be set again.

The option "-x" displays the internal board temperature (in degrees Celsius)

Options:

-tdisplay time
-d display date
-x display board temperature
-s set real time clock
-z set local timezone and daylight savings time flag

timezones:

GMT	-11	Samoa
GMT	-10	Hawaii
GMT	-09	Alaska
GMT	-08	USA Pacific
GMT	-07	USA Mountain
GMT	-06	USA Central
GMT	-05	USA Eastern
GMT	-04	Canada Atlantic
GMT	-03	Brazil
GMT	+00	Greenwich, London
GMT	+01	Berlin, Stockholm, Rome, Paris, Madrid
GMT	+02	Athens, Helsinki, Instanbul, Israel
GMT	+03	Kuwait, Moskau
GMT	+04	Abu Dhabi
GMT	+05	Islamabad
GMT	+06	Dakka
GMT	+07	Bangkok, Jakarta, Hanoi
GMT	+08	Hongkong, Singapore
GMT	+09	Tokio, Osaka, Seoul
GMT	+10	Sydney
GMT	+11	New Caledonia

GMT +12 Auckland, Wellington

example

```
$time
time and date command
temperature: 54.0 C
current timezone: +01
daylight saving time: ON
time: 14:55:20
date: 12/31/00

$time -s
time and date command
current timezone: +01
daylight saving time: ON
time: 14:56:00
date: 12/31/00

input timezone +00 >+01
input daylight saving time
press 'SPACE' to change setting, 'ENTER' to enter
daylight saving time ON
input date MM/DD/YY >12/31/00
input local time HH:MM:SS >14:56:00
```

tp **take picture**

synopsis `tp`

description The command tp takes a picture. The system then switches to frame reproduction, to display the frame stored in memory. (Note: When powered up, the camera always shows the so-called live-video from the CCD sensor) The taken picture is stored in the memory area specified with the command **vd** .

type **type ASCII file**

synopsis `type <path>`

description type lists ASCII files. The filename of the file to be listed is specified as the parameter.

example An example of an ASCII file in the flash EPROM is the command file "autoexec" which is interpreted as soon as the camera is powered up.

```
type fd:\autoexec
```

sh **set shutter value**

synopsis
```
sh <number>
```

description The camera's electronic shutter is set with the command sh.The parameter is a decimal value in microseconds. Please note, that not all shutter values are allowed, depending on the camera model.
Please refer to the camera's technical documentation.

examples

`sh 1000`	select 1 millisecond shutter time
`sh 10000`	select 10 milliseconds shutter time
`sh 1000000`	select 1 second shutter time

Since not all shutter values are available, the command replies with the closest value which could be set.

ver **display VC/RT version**

synopsis
```
ver
```

description This command displays the VC/RT operating system version and release number.

example
```
ver
```

result:

```
print software version
Version Version 5.24 Apr  6 2006
FPGA Version 2006/04/03 11:08:36
SENSOR C4SEN204 CPLD Version: 1
```

vd **set video modes**

synopsis
```
vd [[<option>] <frame number>]
vd [-g <gain>]
```

description The video modes can be changed with vd. The following options are available:

no option	live mode/real frame
-l	live mode/real frame
-d	display memory contents
-g	set gain

Live mode shows the image from the CCD sensor. This mode is equivalent to the function of a standard video camera.

Optionally, a page of the video memory can be selected. The number of video memory pages available may vary, depending on the frame size camera type and the memory size.

different from the VCxx cameras, on the VCxxxx cameras live mode always stores the image in memory. This is valid esp. for vmode(0).

wb **white balance**

synopsis wb

description The command wb performs a white balance for color cameras. It is not available for black-and-white cameras and not for cameras with the serial number of a black-and-white camera and a color sensor as a special option.

this command is only available for color cameras!

Procedure:

1. The user enters wb
2. The shell responds with:

```
Please place white object inside yellow frame
and select a brightness between 100 and 180
Press any key for start and end
```

3. The camera enters the interactive mode and displays the average grey value of the region inside the yellow overlay frame.
4. Place a white or grey (colorless) object (e.g. a piece of paper) under the camera covering the complete area inside the yellow overlay frame
5. Adjust brightness (iris of the lens, illumination) so that the average brightness displayed is between the limits (100 and 180). If the values are higher, the values for RGB might be saturated. If the values are lower, the white balance might be inaccurate.
6. If step 5 is not possible, hit a key to exit the interactive mode. Change the shutter setting with the sh – command and repeat steps 1-5.
7. Press any key to exit the interactive mode. The white balance values are calculated, output on the console, stored as system variables (RED, GREEN, BLUE) and the input color lookup table is programmed.
8. If you type vd after the shell's $-promt to get a live image, you will notice that the tint of the image has changed.

6 The Operating System Functions

6.1 Use of exec()

The operating system call `exec()` can be used to dynamically postload programs from the flash EPROM or MMC/SD-Card to the processor's memory.
The program will only require a few milliseconds to postload, depending on its size. Thus, this is suitable for real-time operations.
Parameters can be passed to the called program, like for C subroutines. When the called program terminates, a return value is returned to the calling program, as usual. After the called program terminates, the calling program is reloaded to memory and processing continues where it was interrupted by the function call.
The entire procedure is quite similar to how C subroutines are called, which is an aid to the user.

The following briefly lists the differences to subroutine techniques.

Dynamic postloading	Subroutine techniques
The function itself is named "main()" It is called by its filename (=subroutine name)	Subroutine can be given any name. Name identical when called
Call the program with the function "**exec**(name,p1,p2,...pn); " p1,p2,...pn are the parameters	Direct call by specifying the program name, e.g. "prog(p1,p2,...pn);"
There are several small programs; each is linked only with the subroutines it requires, shortening linking time	There is one large program, which must be linked with all required subroutines and library functions
Individual (sub-)programs can be replaced quickly and easily, e.g. for testing purposes	The program must always be compiled and linked with the subroutine
Postloading requires CPU time	All subroutines are always available immediately

The following is an **example for a called program**:

```
int main(int p1,int p2,...int pn)
{
}
```

p1,p2,...pn are the parameters passed by exec

Parameters p1, .. pn are restricted to 32bit values (e.g. int, int *, etc.) "long" values (these are 40 bit !!!) are not supported. The maximum number of parameters is 8
The stack size cannot be changed by the linker command file (cc.cmd) for `exec()`

Absolute linked programs are usually loaded starting at memory address 0xZ0200000 (Z=A,E or 8). All user programs including the shell and all absolute linked programs called by exec are loaded this way.

Advanced users may change the *.cmd file to load programs to a different address.

Since of VCRT Release 5.23 it is possible to use relocatable linked programs. The address where these programs will be loaded is determined by the loader at run-time and depends on the memory layout of the VCRT system.

Most programs use initialized variables (string constants, global variables and statics). These variables are initialized to a value which is precalculated at compile-time each time the program is loaded (e.g. by exec).

The following rules must be obeyed:

- **Loading of one program replaces others (e.g. the shell) at the same address**
- **Global variables, statics and string constants don't survive because they are initialized every time loaded.**
- **The stack survives (i.e. local variables) (Because not initialized).**
- **The vcmalloc-area survives (Because not initialized).**
- **The DRAMmalloc area survives, (Because not initialized).**
- **Flash EPROM areas survive (Because not initialized)**

6.2 Use of exec2( ) for starting new tasks

```
int exec2(char * fname, …);
```

The system functions `exec1( )` and `exec2( )` are used to start a new process in the background.

Note: You should not use `exec1( )` anymore, it exists only for compatibility purposes.

Before you execute `exec2( )` you could tune the priority of the task you want to start next with the system variable `TPRIORITY` (default priority is 9)

Furthermore you could also tune the timeslot for the new task with the system variable `TIME_SLICE` (default time slice is 10 ms). A value of 0 for the `TIME_SLICE` variable tags the new task scheduler scheme as FIFO instead of ROUND ROBIN.

The return value of `exec2( )` is either 0 if the new process could not be created or a 32 bit Value representing the task id of the newly started process.

The following is a **sample for a called program**:

```
int main(int p1,...int pn)
{
}
```

p1...pn are the parameters passed by exec2

Parameters p1, .. pn are restricted to 32bit values (e.g. int, int *, etc.) "long" values (these are 40 bit !!!) are not supported. The maximum number of parameters is 2 !
With relocateable code and `exec2( )` you can use a bigger stack size than with `exec( )` because the stack is allocated by the loader as specified in the linker command file (cc.cmd) !

6.3 Use of events

If you don't want to poll for external or internal events, you can use the VCRT event system.

```
void event_connect_to_task(void);
void event_disconnect(void);
int  set_evt(int id);
int  wait(int id, int timeout);
```

If you run your program not as an extra task, your program could directly use the `wait()` function of the event system to wait for a special event without wasting system resources.

You can give a timeout value for `wait()` to make sure it will be terminated within this timeout time frame.

The return values of `wait()`

`1`	the event occurred
`2`	the event has occurred before `wait()` was called
`-1`	indicates a timeout has occurred

If you run your program in the background as an extra task you have to first connect this task to the event system by calling `event_connect_to_task()`function !

If you exit this task you should call `event_disconnect()` to free some memory used by the event system.

The currently available events are listed in `VCRT.H`

```
#define         TIMER           0
#define         MM_CARD         1
#define         IMAGE_READY     4
#define         EXP_READY       5
#define         DHCP_READY      6
#define         TRIG_READY      7
#define         PLC_INT         8
#define         I2C_INT         9
#define         TIMER2          10
#define         DISPLAY_EVT     11
#define         IO_EVT          12
#define         LB_INT          13
#define         COMP1_INT       14
#define         COMP2_INT       15
#define         COMP3_INT       16
```

`TIMER`	is the NULL-event, i.e. only the timeout feature is used
`MM_CARD`	is an event internally used for the media card. Since this event is necessary for media card access, tasks that access the media card MUST connect to the event system by the `event_connect_to_task()`function.
`IMAGE_READY`	signals that an image capture has completed
`EXP_READY`	signals that the exposure of an image has finished. This always happens before the image is stored in memory, i.e. the event `EXP_READY` always comes prior to the event `IMAGE_READY.`
`DHCP_READY`	Used internally for DHCP

TRIG_READY	Event generated by the trigger input (or, if configured by the incremental encoder)
PLC_INT	This event is set, when there is a change on the external PLC inputs
I2C_INT	Used for I2C communication (for SBC4018 only)
TIMER2	Event for TIMER2 (not available for VC20xx smart cameras). Event is set, when TIMER2 counts down to 0.
IO_EVT	This event signals if an IO state change occurred (VC64XX only).
LB_EVT	This event is triggered if an Encoder Lightbarrier change occurred (cameras with encoder Hardware only) .
COMP1_EVT	This event is triggered if an Encoder Comparator1 change occurred (cameras with encoder Hardware only) .
COMP2_EVT	This event is triggered if an Encoder Comparator1 change occurred (cameras with encoder Hardware only) ..
COMP3_EVT	This event is triggered if an Encoder Comparator1 change occurred (cameras with encoder Hardware only) .

If you want to use your own events you should use a free event number not already used in VCRT.H ! This can be done using the system variables USR_EVENT and USR_EVT_LAST. See the system variable chapter for detailed documentation.

To signal an event you must use the `set_evt()` routine - i.e. `set_evt(MY_EVENT);`

All events (currently 0..63) are available for all tasks connected to the event system.

6.4 Use of compressed executeables

If you want to store a program file in the flash device which is too big to fit in there, you can compress the .out file with a special tool called VCZIP (see VC-Download-Support).

The resulting ".cex" file (compressed executeable) will be decompressed and executed automatically when you start the file with its name. Since executables with the same name and the extension .exe or .000 are searched first, be sure to have only the .cex file on the drive.

How to make ZIP files for VC20XX,VC4XXX,VC6XXX and VC7XXX cameras.

Using the VCZIP utility, program files can be compressed to about 40% of the original "*.out" file size. The .cex file gererated for FTP upload is already of its final size. The .msf file generated is about the size of the input .out file, however in flash memory the resulting program file is compressed with the same compression ratio.

Follow these steps in order to compress a linker output "*.out" file:

1) Unzip all files from "vczip.zip" in one folder.
2) Copy the file you want to compress in the folder (example "new.out")
3) call the function `vczip` with the corresponding file name

example: `vczip new`

4) upload the .msf file ("new.msf") to the camera via RS232 or Telnet.
   Alternatively, upload the "new.cex" file into the camera memory using FTP.

5) Either way the newly uploaded file will show the file extension .128
   in the flash memory or .cex on the SD-card

6) start the program as usual, by calling the program name from the shell or
   an autoexec file.

6.5 Overview of the VCRT Library Functions

Wherever necessary, the library functions described below can be linked to any C program.

- memory allocation functions
- flash eprom file functions
- I/O functions (RS232, screen, PLC, Ethernet)
- DRAM access functions
- Functions for processing pixel lists
- video control functions
- rs232 functions
- Flash EPROM access functions
- utilities
- TCP/IP functions **(→ separate documentation)**
- lookuptable functions
- time related functions

6.6 Memory Allocation Functions

Allocation of memory is supported by a series of functions. For the heap space the functions sysmalloc() and sysfree() may be used which very closely resemble the original K & R routines malloc() and free(). The system memory allocation is initialized on power-up. The functions vcmalloc() and sysfree() provided in earlier versions of VC/RT are kept but are based on sysmalloc() and sysfree() using macros.

vcmalloc	user memory allocation
vcfree	user memory release
sysmemfree	returns amount of available user memory
sysmalloc	system memory allocation
sysfree	system memory release

DRAMScreenMalloc	allocate DRAM memory for full screen storage

vcmalloc **user memory allocation (macro)**

synopsis

```
void *vcmalloc(unsigned int size)
```

description

`vcmalloc()` allocates heap memory in the processor's data memory segment. `size` is the size of the requested memory area in words (int=32 bits).

This function returns a pointer to the allocated memory area. If the requested memory is not available as a coherent block, the returned value is the null pointer.

`vcmalloc()` is basically equivalent to the function malloc(), which most systems provide as a runtime library function but its allocation unit is a WORD, not a BYTE.

The use of `malloc()` from the runtime library of the TI cross-development system is also possible. In this case, the memory is allocated from the task's heap. The heapsize must be configured accordingly for the linker command file cc.cmd in this case.

see also vcfree(), sysmalloc()

vcfree **user memory release (macro)**

synopsis

```
void vcfree(void *ptr)
```

description The function `vcfree()` releases the memory allocated by `vcmalloc()` for further use.

`vcfree()` is basically equivalent to the function `free()`, which most systems provide as a runtime library function.

The use of the function `free()` from the runtime library of the TI cross-development system is also possible for heap memory which was allocated with `malloc()`.

example

```
#include <vclib.h>

int *p;
p = (int *)vcmalloc(100);
blrdb(50, p, 0L);
vcfree(p);
```

see also `vcmalloc(), sysmalloc()`

sysmemfree **returns amount of available user memory**

synopsis

```
int sysmemfree(void)
```

description The function `sysmemfree()` returns amount of the available system memory. This can be a useful programming routine, especially in the test phase.

see also vcmalloc(), vcfree()

sysmalloc **system memory allocation**

synopsis

```
void *sysmalloc(unsigned nwords, int type)
```

description

`sysmalloc()` allocates system memory in the processor's SDRAM memory. `nwords` is the size of the requested memory area in words (int=32 bits).

This function returns a pointer to the allocated memory area.

`type` is the type of memory requested. The following tables gives an overview of the various memory types:

Type	Mnemonics	Usage
0	`MTEXT`	Program
1	`MSTACK`	local variables, stack
2	`MDATA`	global variables & heap
3	`MIMAGE`	image data

The reason for this segmentation into 4 different memory spaces is that the DSP is able to keep one page open for each of the 4 different segments. A copy e.g. from stack to data space could then be performed at the highest possible speed without unnecessary page access cycles (RAS) for the memory. At the same time the text segment could be accessed for executable machine code.

the memory-`type` is currently not used!

`sysmalloc()` tries to return a pointer to the requested type and size of memory. It is allowed to return a pointer to a different memory type in case the requested type has not enough space. If the requested memory is no longer available as a coherent block, then the function will return the null pointer.

see also vcfree(), sysfree()

sysfree **system memory release**

synopsis

```
void sysfree(void *ap)
```

description The function `sysfree()` releases the memory allocated by `sysmalloc()` for further use by the operating system.

example

```
#include <vcrt.h>

int *p;
p = (int *)sysmalloc(1000,2);
blrdb(50, p, 0L);
sysfree(p);
```

see also `vcfree(), sysmalloc()`

DRAMScreenMalloc **allocate DRAM memory for full screen storage (macro)**

synopsis

```
U8 *DRAMScreenMalloc(void)
```

description The function `DRAMScreenMalloc()` allocates SDRAM memory for one screen of video display + 1024 bytes. It returns the start address of the allocated memory block. This start address may be used to instruct the video controller to display the memory area on the video monitor. Be sure to align the address to a multiple of 1024 for this purpose.

The macro can be found in `macros.h`. `NEW_IMAGE_VAR` must be defined for his macro to output a `U8` address, otherwise it returns `I32` as result.

This function can also be used to allocate overlay memory.

example

```
#define NEW_IMAGE_VAR
#include <macros.h>

U8 * addr = DRAMScreenMalloc();
setvar(DISP_START, (addr+1024) & ~1023);
```

6.7 General I/O Functions

Files and I/O devices are accessed by means of generalized I/O functions. This is a new feature for VC/RT 5.0x with respect to earlier versions.

We strongly recommend the use of these functions instead of direct functions (like search, fnaddr, etc.). The latter will be kept for a while for compatibility purposes.

The following functions are available:

io_fopen	open a device, get file pointer
io_fclose	close device
io_read	read from device
io_write	write to device
io_ioctl	control function
io_fgetc	get character from device
io_fputc	put character to device
io_fseek	set file position
io_get_handle	get a pointer to the default standard I/O stream
io_pipe_install	Install a pipe device

The standard procedure for file operations is as follows:

```
io_fopen()

/* ... one or more file operations ... */

io_fclose()
```

The operation **io_fopen**() locks a file for access from other tasks depending on the access mode and allocates some buffers for that file. **io_fclose**() frees the memory used and unlocks the file so that it may be used subsequently by another task. For this reason we recommend using the function **io_fclose**() immediately when access to the file is no longer necessary.

The following devices are available:

Name	Device Type	Description
fd:	block	Flash EPROM file device
md:	block	Multi Media or SD-card device
ittya:	char	Serial communication channel for serial VC20xx cameras / not available for VC40xx cameras
kbd:	char	Serial keyboard channel for VC20xx cameras / Serial channel for keyboard and other devices for VC40xx cameras
telnet:	char	Telnet communication channel for all Ethernet cameras
socket:	network	Internal network channel. Do not use !
dbg:	pipe	Debug pipe, used by kl shell command
t0:	pipe	Internal Pipe for telnet communication. Do not use !
t1:	pipe	Internal Pipe for telnet communication. Do not use !
x1:	pipe	Internal Pipe for decompression. Do not use !

char and pipe devices are not buffered, block devices are buffered (standard buffer size: 4096 bytes)

The following restrictions apply:

Drive	Access Mode	Operation
fd:	Read	Unlimited number of read accesses to same file
	Write	Access to only 1 file in total for write
md:	Read	Unlimited number of read accesses to same file
	Write	Access to file is locked for other tasks An unlimited number of files may be open for write
“pipe”:	Read	Access to only 1 pipe in total per devicename
	Write	Access to only 1 pipe in total per devicename

For special I/O operations the function **io_ioctl**() may be used. Here, a drivename, path or file must be opened with **io_fopen**() and **mode="c".** Then the **io_ioctl**() is performed. Finally the function **io_fclose**() must be called.

io_fopen **open a device, get file pointer**

synopsis

```
FILE *io_fopen(char *path, char *mode)
```

description

The function **io_fopen**() opens a device / file / directory with the pathname given by path.

It returns the filepointer if successful or NULL if not.

It is possible to open the device with the following mode-strings:

mode =	"r"	read
	"w"	write
	"c"	control
	"a"	append

io_fclose **close a device**

synopsis

```
int io_fclose(FILE *fp)
```

description

The function **io_fclose**() closes a device / file / directory previously opened with io_fopen.
The function returns 0 for successful operation or otherwise an error number, which depends on the driver for the selected device.

io_read **read from device**

synopsis
```
int io_read(FILE *fp, char *buf, int cnt)
```

description The function **io_read**() reads from a device / file previously opened with io_fopen.

`cnt` is the number of bytes,
`buf` is a pointer to a buffer to store the data.

The return value of the function is the number of bytes transferred if successful or else -1.

io_write **write to device**

synopsis
```
int io_write(FILE *fp, char *buf, int cnt)
```

description The function **io_write**() writes to a device / file previously opened with with io_fopen. `cnt` is the number of bytes, buf is a pointer to a buffer of data to be written. The return value of the function is the number of bytes transferred if successful or else -1.

io_ioctl **I/O control**

synopsis
```
int io_ioctl(FILE *fp, unsigned cmd, void
*param)
```

description The function **io_ioctl**() is used for various device control functions.

cmd is a command code to request a certain function, param is a pointer to a variable or struct, where information may be passed from the calling routine to the function or vice versa.

Here is a list of available functions

device	cmd	function	param
ittya:, kbd:	IO_BAUD_SET	set baud rate	&baud
	IO_BAUD_GET	get baud rate	&baud
	IO_RTS_SET	set RTS to 1 *)	NULL
	IO_RTS_CLR	set RTS to 0 *)	NULL
	IO_IOCTL_SERIAL_GET_FLAGS	get communication flags	&flags
	IO_IOCTL_SERIAL_SET_FLAGS	get communication flags	&flags
fd:	IO_PACK	pack	&result
	IO_ERASE	erase	&result
	IO_READDIR	read directory	READDIR
	IO_CHKSYS	check system	NULL
	IO_DEL	delete file	NULL
	IO_REMAIN	remaining device space	&size
md:	IO_READDIR	read directory	READDIR
	IO_DEL	delete file	NULL
	IO_MKDIR	make directory	NULL
	IO_REMAIN	remaining device space	&size
“pipe”:	IO_PIPE_CHMOD	change mode	&mode
	IO_PIPE_CHSIZ	change size and reset pipe	&size
	IO_PIPE_RDFLAGS	read out mode flags	&flags
	IO_PIPE_GETCOUNT	get number of characters	&num
	IO_PIPE_SIZE	size of pipe	&size

*) For cameras with serial hardware handshake only (VC20xx)

io_fgetc **get character from device**

synopsis
```
int io_fgetc(FILE *fp)
```

description The function **io_fgetc**() inputs a character from the **device** fp. If an End-Of-File condition is encountered, -1 is output instead of a character

io_fputc **output character to device**

synopsis
```
int io_fputc(int c, FILE *fp)
```

description The function **io_fputc**() outputs a character to the device fp.

The return value of the function is equal to the character c written or a negative error condition.

io_fseek **set the file position**

synopsis
```
int io_fseek(FILE *fp, int offset, unsigned
start_from)
```

description The function **io_fseek**() positions the read-filepointer to the position specified with offset.

On success the function returns 0.

The following values are possible for start_from:

IO_SEEK_SET	offset
IO_SEEK_CUR	current_position + offset
IO_SEEK_END	file_size + offset

io_get_handle **get a pointer to the default standard I/O stream**

synopsis
```
FILE *io_get_handle(unsigned stdio_type)
```

description The function **io_get_handle**() returns a pointer to the default standard I/O stream.

If unsuccessful, NULL is returned.

stdio_type may be any of the following values:

```
IO_STDIN
IO_STDOUT
IO_STDERR
```

io_pipe_install **install a pipe device**

synopsis
```
I32 *io_pipe_install(char *name, U32 size)
```

description The function **io_pipe_install**() installs a pipe device with name and size in bytes.

example
```
io_pipe_install("pipe0:", 1000);
```

It is possible to install an arbitrary number of pipes with different names. Do not use a name more than once !
A pipe can only be opened once for writing and once for reading. Trying to open a pipe a second time for a given mode will return an error code for **io_open**().

6.8 Program execution

exec	load and execute a program
exec2	load/execute as a parallel task

exec **Load and execute a program**

synopsis

```
exec (char *path, p1,p2, ... , pn)
```

description

With the function `exec()`, programs (subroutines) are loaded from the Flash EPROM or from the media card / SD-card to the SDRAM memory of the DSP and executed. First, the path (`char * path`) is used to search for the file. If the file is found, the loading and starting process begins. If the file is not found, a soft reset is invoked. Thus, make sure the file can always be found (e.g. with the function `io_fopen()`).

Up to 8 (int) parameters can be passed to the program, as p1, p2, ... , pn.
All parameters are restricted to 32 bit values (e.g. int, int *) "long"-values are not supported, as they are 40 bit.

When the program terminates, the calling program will automatically be loaded back into memory. Integer (32 bit) values can be returned to the calling program.

The following applies for the called program:
Its name is:

```
int main(int p1, int p2, ... , int pn)
{
}
```

where p1,p2,...pn are the parameters passed over from exec.

The function exec() can be used to dynamically postload subroutines from a main program. Subroutines loaded via exec() may be nested. Naturally, the size of the stack limits the level to which subroutines can be nested.

If many parameters must be passed to the function called by exec(), a pointer to a struct on the stack or on the heap may pe passed alternatively. Keep in mind that pointers use 32 bits. They will therefore fit easily in the space of an int (32 bits). The called program may also modify the struct's items.

Do not try to pass string constants to a function called by exec(). Since string constants are represented by a pointer to initialized memory areas, the string information may be lost (overwritten) when the function is called.
If you have to pass strings, then copy them to a local variable first and pass the local variable or it's address instead.

example **DO NOT !!!** `exec("myprog","this string should not be here")`

exec2 **Load and execute a program as a parallel task**

synopsis

```
int id=exec2 (char *path, p1,p2, ... , pn)
```

description

With the function `exec2()`, programs (subroutines) are loaded from the flash EPROM to the SDRAM memory of the DSP and executed as an extra task. First, the path (`char * path`) is used to search for the file. If the file is found, the loading and starting process begins. If the file is not found, a soft reset is invoked.
Thus, make sure the file can always be found (e.g. with the function `io_fopen()`). The return value is 0 in case the new task could not be started or a int value representing the task id. **Up to 2 (int) parameters** can be passed to the program, as p1, ... , pn.

6.9 I/O Functions

pstr	Output a string via the serial interface
print	Formatted output of text and variables
sprint	Formatted output of text and variables to a string
hextoi	convert hexadecimal value string to integer
setRTS	set RTS signal (macro)
resRTS	reset RTS signal (macro)
setPLCn	set PLC signal (macro)
resPLCn	reset PLC signal (macro)
outPLC	output value to PLC
inPLC	input value from PLC (macro)

pstr **Output a string via the serial interface**

synopsis
```
void pstr(char *str)
```

description This function outputs the string specified by the pointer str via the serial interface. This function differs from the function print() in that **pstr**() must not contain format control characters such as %.
For the ASCII character LF (0x0a or '\n'), a combination of CR (0x0d or '\r') and LF is output.

print **Formatted output of text and variables**

synopsis
```
void print(char *format, ...)
```

description This function is a full-featured version of the standard function **printf**().

The following is a list of formats supported:

format-string	remark
%d	decimal number / 32 bits
%u	unsigned decimal number / 32 bits
%x, %X	hex number / 32 bits
%o	octal number / 32 bits
%ld, %lu, %lx, %lo	same as above for 40 bit long values
%hd, %hu, %hx, %ho	same as above for 16 bit short values
%c	character
%s	string
%p	pointer / 32 bits
%n	number of arguments
%f	floating-point (double)
%e	floating-point (double)
%g	not implemented
*	variable number of arguments

The text and variables are output via the serial interface, resp. Ethernet port..

Since the argument list is variable (...), print() only works properly if the correct prototype is included in the user program. This can be done, for example, by adding the following line:

```
#include <vcrt.h>
```

see also sprint(), pstr()

sprint **Formatted output of text and variables to a string**

synopsis

```
void sprint(char *s, char *format, ...)
```

description The function sprint() is equivalent to the function **print**(), however the output is directed to the passed string s.

This can be used, for example, to prepare the output of data on the screen.

Since the argument list is variable (...), sprint() only works properly if the correct prototype is included in the user program. This can be done, for example, by adding a line

```
#include <vcrt.h>
```

see also print()

hextoi **convert hex value string to integer**

synopsis

```
int hextoi(char *s)
```

description The '\0' terminated character string s containing the hexadecimal value is passed to the function. The function then converts it to an integer value.

setRTS **set RTS signal (macro)**

synopsis

```
void setRTS(void)
```

description This macro sets the RTS output of the V24 (RS232) interface to a positive voltage. This allows communication, i.e. characters are allowed to be sent to the camera from the connected computer. Make sure that the host computer is switched to "hardware handshake" if you want to use this feature

Hardware handshake is available only for the serial version of the VC20xx smart cameras.

resRTS **reset RTS signal (macro)**

synopsis
```
void resRTS(void)
```

description This macro resets the RTS output of the V24 (RS232) interface to a negative voltage. This shuts down communication, i.e. characters are not allowed to be sent to the camera from the connected computer. Make sure that the host computer is switched to "hardware handshake" if you want to use this feature

Hardware handshake is available only for the serial version of the VC20xx smart cameras.

setPLCn **set PLC signal (macro)**

synopsis
```
void setPLCn(void)
```

description This macro sets the PLC signal no. n, so that current is flowing through the corresponding output. The signal will have a positive voltage.

example
```
setPLC0();      /* switch on output 0   */
setPLC1();      /* switch on output 1   */
setPLC2();      /* switch on output 2   */
setPLC3();      /* switch on output 3   */
```

resPLCn **reset PLC signal (macro)**

synopsis
```
void resPLCn(void)
```

description This macro resets the PLC signal no. n, so that no current is flowing to the corresponding output. The signal will be high-impedance.

example
```
resPLC0();      /* switch off output 0       */
resPLC1();      /* switch off output 1       */
resPLC2();      /* switch off output 2       */
resPLC3();      /* switch off output 3       */
```

outPLC **output value to PLC**

synopsis
```
void outPLC(value)
```

description This function outputs value to the PLC. The function also writes the value to the system variable PLCOUT where the state of the output signals can be monitored at any time. Bits 0 to 3 of value will set the corresponding output signals.

If more than 4 outputs are necessary, Beckhoff I/O modules may easily be connected to VC smart cameras. See the separate documentation for usage.

inPLC **input value from PLC (macro)**

synopsis

```
int inPLC(void)
```

description

This macro inputs the status of the PLC input signals. Bits 0 to 3 indicate the status of each individual PLC input. The remaining bits are always zero. A zero on one of the input bits means that there is current flowing through the corresponding PLC input. If there is no voltage on the input, the bit will be 1.

The status of the PLC input bits can also be monitored using the system variable PLCIN. This variable, however, features an additional status bit (bit #4) which indicates failure of the PLC I/O processor when set to 1.

If more than 4 inputs are necessary, Beckhoff I/O modules may easily be connected to VC smart cameras. See the separate documentation for usage.

6.10 Video Control Functions

capture_request	put request for image capture into capture queue
capture_request2	capture_request with encoder support
cancel_capture_rq	stop capture request
vmode	Set video modes
tpict	Picture taking function
tpp	Picture taking function / progressive scan
tpstart	Picture taking function / progressive scan
tpwait	Wait for completion of picture taking function / progressive scan
tenable	Trigger enable for interrupt driven image acquisition
trdy	Check the status of the picture taking function / external trigger mode
shutter	select shutter speed
SET_trig_lossy	select "lossy" external trigger mode
SET_trig_sticky	select "sticky" external trigger mode

capture_request **put request for image capture into capture queue**

synopsis

```
int capture_request (int exp, int gain,
                                  int *start, int mode)
```

description

This is the most basic function for capturing an image on which all other functions in this chapter like tpict or tpp are based. With this function, the user is able to achieve the best performance for the video capture process.

It is possible for the image acquisition hardware, especially for the sensor to process more than one image capture requests in parallel. It can read out one image and transfer it to memory while exposing another one. So, the maximum frame rate can be achieved. Of course there are some limitations:

The maximum frame rate can only be achieved if the exposure time is less than the read-out time. Otherwise, the maximum frame rate is determined by the exposure time.

Exposure starts when the time left for read-out equals the exposure time or is less. If the image acquisition is triggered by software (mode=0), it always starts as soon as possible. If the image is triggered externally (mode=1), the user may choose the trigger to be "lossy" (`SET_trig_lossy()`) or "sticky" (`SET_trig_sticky()`). In the first case the trigger will be lost, if it comes too early, in the latter case, it will be stored until image acquisition is possible.

With this function, complete control and tracking individual images is possible. The following parameters may be set for individual images:

`exp` exposure time in units of EXUNIT msecs

`gain`	gain setting for ADC
`start`	start address for image storage
`mode`	internal / external trigger mode (mode=0 : int., mode=1 : ext.) binning (mode=8 : binning enabled)

Exposure time is calculated according to the following formula:

```
Exptime[µsec]=(exp +
 getvar(XSG)/getvar(TOTAL))*getvar(EXUNIT)
```

So, exp=0 means a shutter time of approximately 30 msecs for a VC4038. Shutter times may be quite large, e.g. several seconds. Please note, that with shutter times above 1 sec individual pixels may feature large amounts of spot noise, those pixels may even be fully saturated. This is normal and no reason for return of equipment. Use appropriate filtering to reduce this kind of noise.

Gain is calculated according to the following formula:

realgain[dB] = 6 + (32*gain/256) accuracy: +/- 1dB

The amplification of the PGA may then be calculated with the following formula:

amplification = 10^(realgain/20)

For large differences in gain from one picture to the next, the ADC may take some time to track the black level. If this is a problem, you should insert one picture for adjustment.

Be sure that you have allocated enough memory at address start for the image to be stored.

Mode=1 means external trigger. If the corresponding image is processed, the system waits for the external trigger to start ackquisition. The system may wait indefinitely in this state if no trigger is received. If this state needs to be cancelled without external trigger, the function `cancel_capture_rq()` may be used.

Mode=8 activates factor 2 binning (for cameras featuring binning), i.e. the vertical number of lines is reduced to half and the sensitivity is doubled. Binning is a special feature of CCD sensors, where consequtive line pairs are added on the sensor.

The capture requests are put into a queue of 20 slots. As long as slots are available, a call of `capture_request()` returns immediately regardless if the picture is taken without delay or the request is just stored in the queue.

If the queue is full, the function will return 0. No request is stored.

When the request is stored, the function returns a non-zero token or tracking number for this request. This number may be used to poll the system variables `EXPOSING`, `STORING` and `IMGREADY`, where the tracking numbers of the images requested in the different states are shown.

It is not allowed to call this function in live mode (vmode(0), vmode(2), vmode(4), vmode(6)). This is not checked !

cancel_capture_rq **cancel capture request**

synopsis

```
int cancel_capture_rq(void)
```

description

Sometimes it is necessary to abort the currently active capture request queue. This is e.g. the case, when a capture request has been issued with an external trigger, but the trigger signal does not come.

`cancel_capture_rq()` aborts the capture request queue and resets the capture hardware.

The function first checks if a capture transfer is currently active (i.e. data being captured from CCD previously is transferred into main memory) If so, the function returns 1 and does not perform a cancel operation. If not, the cancel is done immediately and will return 0. Execution time: approx. 1 msec, when successful.

`cancel_capture_rq()` does the following:

- stop live mode and set `IMODE` to 1
- set VSTAT to 0
- reset capture hardware
- clear capture queue
- initialize capture driver

example

```
while(cancel_capture_rq() == 1);
```

see also `capture_request()`

vmode **Set video modes**

synopsis

```
void vmode(int mode)
```

description

This function changes the modes for the video controller.

The settings are made according to the following table:

mode	meaning	IMODE	OVLY_ACTIVE
0	live video + cyclic image acquisition	0	0
1	display of the video memory (stills)	1	0
2	live video + cyclic image acquisition	0	0
3	display of the video memory (stills)	1	0
4	like 0 but including overlay display	0	1
5	like 1 but including overlay display	1	1
6	like 2 but including overlay display	0	1
7	like 3 but including overlay display	1	1

Other values for mode are not defined.

The setting of the system variables determines the location and format of the display (mode 1, 3, 5, 7) and how the frame is stored (mode 0, 2, 4, 6).

The function changes the value of the system variables IMODE and OVLY_ACTIVE (see table). Changes of the video mode come into effect after the completion of the next frame.

tpict **Picture taking function**

synopsis

```
void tpict(void)
```

description

This function takes a picture. The function waits in a loop until the entire picture is in memory. This function was implemented to provide a "compatibility mode" to the `tpict()` function of cameras not equipped with progressive scan sensor.
This function does not, however, completely support the special progressive scan features. It is therefore recommended to use the functions `capture_request()` or `tpp()`, whenever the special progressive scan features are needed.

The current setting of the system variables determines the location and format of the stored picture in memory.

tpict() changes the video mode. After this function is called, the system switches to still frame (vmode=1). If overlay is active, the system switches to still frames with overlay (vmode=5).

The function changes the value of the system variable `IMODE` to 1.

tpp **Picture taking function / progressive scan**

synopsis

```
int tpp(void)
```

description

This function takes a picture in progressive scan mode. This means, that the sensor starts with image integration without any delay. The exposure time is determined by the selected shutter speed which can be controlled with the `shutter()` function.
After the image integration, the information is transferred to the DRAM. The sensor always works in full frame mode, i.e. there are no half images.
The function waits in a loop until the entire picture is in memory. It is not allowed to call `tpp()` in all video modes. See the following table for allowed video modes:

vmode setting	description	use of tpp()
vmode(0)	live video storage	not allowed
vmode(1)	still video	allowed
vmode(2)	live video storage	not allowed
vmode(3)	still video	allowed
vmode(4)	vmode(0) + overlay	not allowed
vmode(5)	vmode(1) + overlay	allowed
vmode(6)	vmode(2) + overlay	not allowed
vmode(7)	vmode(3) + overlay	allowed

If `tpp()` is called in a video mode for which it is not allowed, it returns -1 and no picture is taken. If it is necessary, to take a picture while being in one of the not allowed video modes, the function `tpict()` may be used. This, however, means that the immediate triggering of the progressive scan sensor cannot be used.

tpp() **does not change the video mode**.

The following example shows the use of `tpp()` with external trigger.

example

```
vmode(1);                       /* still mode on        */
tpwait();                       /* wait for still mode  */

while(inPLC()&0x01) != 0);      /* wait for trigger     */
tpp();                          /* take picture         */
```

Using this function does not support parallel processing (exposing while storing the image). For maximum performance, the function `capture_request()` is recommended.

tpstart **Picture taking function / progressive scan**

synopsis
```
int tpstart(void)
```

description This function is quite similar to the function tpp(). The only difference is that it does not wait for the completion of the image taking process.

Using this function does not support parallel processing (exposing while storing the image). For maximum performance, the function `capture_request()` is recommended.

tpwait **Wait for completion of picture taking function (macro)**

synopsis
```
void tpwait(void)
```

description This function is used to make sure, that an image taking process, started with `tpstart()` is completed. If so, the function immediately returns, if not, the function waits in a loop.

tenable **Trigger enable for interrupt driven image acquisition**

synopsis
```
int tenable(void)
```

description this function resembles the `tpp()` function, except for the fact that the start of the image integration is triggered by the external trigger input. An image can only be triggered externally, if `tenable()` has been called before.
A call of `tenable()` activates the acquisition of one image only. After the call of `tenable()` the function returns to the caller, so processing can be done in parallel to image acquisition. Of course, it makes no sense to process an image which might change due to an external trigger, but the processing of a previously stored image is possible.

For details of the image-taking process, please refer to the documentation of the `tpp()` function.

if tenable() is called in a video mode for which it is not allowed, it returns -1 and the picture-taking is not enabled.

Please do not change the video mode after tenable() has been called and before the image has been successfully stored in memory.

Using this function does not support parallel processing (exposing while storing the image). For maximum performance, the function `capture_request()` is recommended.

trdy **Check the status of the picture taking function**

synopsis

```
int trdy(void)
```

description

This function is used to check, if an image taking process, started with `tenable()` is completed. If so, the function returns 1, if not, the function returns 0.

example

```
vmode(1);          /* still mode on                    */
tpwait();          /* wait for still mode              */

tenable();         /* now wait for external trigger    */
while(!trdy());    /* wait for completion              */
```

shutter select shutter speed

synopsis

```
long shutter(long stime)
```

description

This function selects the shutter speed for the CCD sensor.The shutter speed is specified with the value stime in microseconds.
The shutter speed of the sensor will be set to a possible value close to the one specified. The function will return the exact shutter speed selected in microseconds. The possible shutter values range from approx. 90 msec to several seconds depending on the CCD sensors.

With shutter times above 1 sec individual pixels may feature large amounts of spot noise, those pixels may even be fully saturated. This is normal and no reason for return of equipment. Use appropriate filtering to reduce this kind of noise

SET_trig_lossy **select "lossy" external trigger mode (macro)**

synopsis

```
void SET_trig_lossy(void)
```

description

If the external trigger mode for the image acquisition is selected, there is an error condition if the trigger signal is set during the ackquisition time of the previous page. In this case the user may choose to lose the trigger information (`SET_trig_lossy()`) or store it until image acquisition becomes possible (`SET_trig_sticky()`).

SET_trig_sticky **select "sticky" external trigger mode (macro)**

synopsis

```
void SET_trig_sticky(void)
```

description

If the external trigger mode for the image acquisition is selected, there is an error condition if the trigger signal is set during the ackquisition time of the previous image. In this case the user may choose to lose the trigger information (`SET_trig_lossy()`) or store it until image acquisition becomes possible (`SET_trig_sticky()`).

6.11 RS232 (V24) Basic Functions

rs232snd, putchar	output a character/serial interface
rs232rcv, getchar	read a character/serial interface
sbready	send buffer ready/serial interface
rbready	receive buffer ready/serial interface
rbempty	receive buffer empty/serial interface
setbaud	set baudrate for serial interface
kbdrcv	read a character/keyboard
kbready	receive buffer ready/keyboard

rs232snd, putchar **Output a character/serial interface**

synopsis

```
void rs232snd(char c)
void putchar(char c)
```

description

This function outputs one buffered ASCII character via the serial interface (STDOUT). If the send buffer is not full, the ASCII character is buffered and program control returns to the calling program. Otherwise, this function waits until there is room in the buffer, buffers the character and then returns to the calling program. Waiting for available buffer space does not consume CPU time.

The buffer is read in the background by an interrupt routine. The character is transferred via the serial interface as a background process.

The send buffer can hold a maximum of 256 characters.

The character output is done using the standard serial device driver. This performs a LF to CR/LF conversion as well as XON/XOFF and hardware handshake. The behaviour of the device driver can be controlled using the function `io_ioctl()`to change the IO-flags of the driver.
The default mode for the serial device driver is LF to CR/LF conversion – no handshake.

see also `rs232rcv(), sbready()`

rs232rcv, getchar **Read a character/serial interface**

synopsis

```
char rs232rcv(void)
char getchar(void)
```

description

This function reads one buffered ASCII character from the serial interface (STDIN). A background interrupt routine writes the character to the buffer. Characters will be lost if the background buffer overflows!

The function rs232rcv() first determines if there is a character in the buffer. If not, it waits until this is the case. The character is then removed from the buffer and handed over to the calling program as a return value. Waiting for a character does not consume CPU time.

The receive buffer can hold a maximum of 256 characters.

The character input is done using the standard serial device driver. This performs XON/XOFF and hardware handshake. The behaviour of the device driver can be controlled using the function `io_ioctl()`to change the IO-flags of the driver.
The default mode for the serial device driver is no handshake.

see also `rs232snd(), rbready()`

sbready **send buffer ready/serial interface**

synopsis
```
int sbready(void)
```

description This function returns the number of available buffer places for the send buffer of the serial interface. If the return value is 0, no space is available and a character output with `rs232snd()` will wait until space gets available.

see also `rs232snd(), sbfull()`

rbready **receive buffer ready/serial interface**

synopsis
```
int rbready(void)
```

description This function returns the number of characters stored in the receive buffer of the serial interface. If the return value is 0, no character is available and a character input with `rs232rcv()` will "hang" until a character becomes available.

buffer space for this function is always 1 character for reasons of compatibility.

see also `rs232rcv(), rbempty()`

setbaud	**set baudrate for serial interface**
synopsis	`void setbaud(long baudrate)`
description	The function sets the hardware baudrate clock to the specified value.
example	`setbaud(9600L) /*set baudrate to 9600baud*/`

kbdrcv	**Read a character/keypad**
synopsis	`char kbdrcv(void)`
description	This function reads one buffered ASCII character from the keypad VCSKB. A background interrupt routine writes the character to the buffer. Characters will be lost if the background buffer overflows! The function `kbdrcv()` first determines if there is a character in the buffer. If not, it waits until this is the case. The character is then removed from the buffer and handed over to the calling program as a return value. Waiting for a character does not consume CPU time. The receive buffer can hold a maximum of 64 characters. The character input is done using the standard serial device driver. This performs XON/XOFF and hardware handshake. The behaviour of the device driver can be controlled using the function `io_ioctl()`to change the IO-flags of the driver. The default mode for the serial device driver is no handshake.

kbready	**receive buffer ready/keyboard**
synopsis	`int kbready(void)`
description	This function returns the number of characters stored in the receive buffer of the serial interface. If the return value is 0, no character is available and a character input with `rs232rcv()` will "hang" until a character gets available.
see also	`kbdrcv(), rbready()`

6.12 Utility Functions

getvar	Read system variable (macro)
setvar	Write system variable (macro)
getlvar	Read system variable (long, macro)
setlvar	Write system variable (long, macro)
getfvar	Read system variable (float, macro)
setfvar	Write system variable (float, macro)
getstptr	Read stack pointer
getdp	Read data pointer
getbss	Read start of bss

getvar **Read system variable**

synopsis
```
int getvar(int var)
```

description The function `getvar()` reads the value of a system variable. `var` is usually a system variable from the file `sysvar.h`

example
```
#include <sysvar.h>
int mode;
mode = getvar(IMODE);   /* get video mode */
```

setvar **Write system variable**

synopsis
```
void setvar(var, int x)
```

description The function `setvar()` changes the value of a system variable. `var` is usually a system variable from the file `sysvar.h`, `x` is the value to be written.

example
```
#include <sysvar.h>
setvar(DISP_ACTIVE,0);  /* disable video refresh     */
```

getlvar **Read system variable (long)**

synopsis
```
long getlvar(int var)
```

description The function `getlvar()` reads the value of a long system variable (40 bits). `var` is usually a system variable from the file `sysvar.h`

setlvar **Write system variable (long)**

synopsis
```
void setlvar(int var, long x)
```

description The function `setlvar()` changes the value of a long system variable (40 bits). `var` is usually a system variable from the file `sysvar.h`, `x` is the value to be written.

getfvar **Read system variable (float)**

synopsis
```
float getfvar(int var)
```

description The function `getfvar()` reads the value of a float system variable. `var` is usually a system variable from the file `sysvar.h`

setfvar **Write system variable (long)**

synopsis
```
void setfvar(int var, float x)
```

description The function `setlvar()` changes the value of a float system variable. `var` is usually a system variable from the file `sysvar.h`, `x` is the float value to be written.

getstptr **Read stack pointer**

synopsis
```
int getstptr(void)
```

description The function `getstptr()` reads the current value of the stack pointer. This can be useful when debugging programs.

getdp **Read data pointer**

synopsis
```
int getdp(void)
```

description The function `getdp()` reads the current value of the data pointer. This can be useful when debugging programs.

getbss **read start of bss**

synopsis
```
int getbss(void)
```

description The function `getbss()` reads the start of the bss space to a C program. This can be useful when debugging programs.

6.13 Lookup Table Functions for Video Display and Overlay

set_overlay_bit	assign a color to an overlay bitplane
set_translucent	assign a color to a translucent overlay table
set_ovlmask	set overlay mask register
init_LUT	init image data LUT to black-and-white display
init_LUT_gamma	Initialize output LUT using gamma-correction
init_color_lut	Initialize input LUT for color camera

set_overlay_bit **assign a color to an overlay bitplane**

synopsis

```
int set_overlay_bit(int bit, int r, int g, int b)
```

description

This function programs the overlay lookuptable. A color given by (r,g,b) can be assigned to the bitplane given by bit.

```
r,g,b ∈ [0,255]
bit ∈ [2,7]
```

6 overlay bit planes (bit=2 .. bit=7) are available for overlay graphics. bit=0 and bit=1 are reserved for translucent overlay graphics.
Higher bitnumbers have priority over lower ones, i.e. whenever a bit is set in n overlay byte, lower number bits of this bytes are "don't care". This rule also applies to the translucent bits 0 and 1, i.e. whenever at least one of the bits 2..7 is set, the overlay pixel is no longer translucent.

The function returns -1 if bit is out of range, else 0.

example

```
image a = {0L, 16, 16, 768};
a.st = (long)getvar(OVLY_START);

markerd(&a,8);    /* draw marker    */
set_overlay_bit(3,0,255,0);   /* green          */
```

set_translucent **assign a color to a translucent overlay table**

synopsis

```
void set_translucent(int table, int r, int g, int b)
```

description

This function programs the overlay lookuptable. A color given by (r,g,b) can be assigned to the translucent table given by table .

```
r,g,b ∈ [0,255]
table ∈ [1,3]
```

3 translucent tables (table=1 .. table=3) are available. The function programs the overlay lookuptable such that it multiplies the upper 6 bits of image data with the color value given by (r,g,b) (The value is then scaled down to 8 bits). The image modifed with this kind of translucent table will look as if it was viewed through a piece of colored glass.

bits 0 and 1 in overlay memory are used to indicate if a given pixel should be modified with on of the 3 translucent tables:

byte value	function
0	no translucent display
1	table no. 1
2	table no. 2
3	table no. 3
> 3	non translucent overlay has priority over translucent table

The function returns -1 if table is out of range, else 0.

example

```
image a = {0L, 16, 16, 768};

a.st = (long)getvar(OVLY_START);
set(&a,1);              /* set to 1       */

set_translucent(1,0,255,255);      /* cyan transl.   */
```

set_ovlmask **set overlay mask register**

synopsis `void set_ovlmask(int mask)`

description This function programs the overlay mask register. A value of `mask`=255 (`0xff`) enables all 8 overlay bitplanes. A value of `mask`=`0` disables all overlay bitplanes. Since in this case the overlay is completely inactive, the function disables also the transfer of video data into the refresh memory by writing a 0 to the system variable `OVLY_ACTIVE`.
Writing a value ≠ 0 to the mask registers with this function will activate the transfer by writing a 1 to `OVLY_ACTIVE`.
The value of `mask` is written to the system variable `OVL_MASK`.

The function `set_ovlmask()` changes the system variables `OVL_MASK` and `OVLY_ACTIVE`.

init_LUT **init image data LUT to black-and-white display**

synopsis `void init_LUT(void)`

description This function programs the image data lookuptable for black-and-white display.

init_LUT_gamma **init image output LUT using gamma correction**

synopsis `void` **`init_LUT_gamma`**`(float gamma)`

description This function programs the image output lookuptable (output LUT) for black-and-white / color display using gamma correction.

Gamma correction is a non-linear function used in order to compensate for display monitor non-linearities.

The following formula is applied:

X' = X ^ gamma , where X may be any of R,G,B

Higher values for gamma tend to increase contrast while at the same time low grey values (dark areas) may not be distinguishable.
Lower values decrease contrast and dark areas may be better differentiated.

The standard value for gamma is 0.45 (according to various video standards).
We recommend a value of 0.6 .
Of course, the best value depends on the chosen monitor and its settings (like brightness and contrast) and may be found using some experimentation.

see also `init_LUT()`

init_color_lut	**initialize color input LUT**
synopsis	`void init_color_lut(I32 red, I32 green, I32 blue)`
description	This function programs the **hardware** input color lookup-table to a linear mapping between input and output. The mappings for the red, green and blue channels can be programmed to a different slope, which is a useful feature for adjusting the whitebalance of the camera. Slope values for `red`, `green` and `blue` can be used to amplify each channel (value > 1024) or attenuate the channel (value < 1024). A value of 1024 will result in an identity transform. 9 bits are used for the input of the LUT, 8 bits for the output, so there is enough head-room for some amplification. For the whitebalance adjustment, we recommend to leave the channel with the maximum intensity at the identity transform, the other two channels should be amplified by appropriate factors. The possible range for `red`, `greeen` and `blue` is [0.. 32768] equivalent to amplification factors between 0 and 32.
side effects	The function changes the values of the system variables `RED`, `GREEN` and `BLUE`.
memory	`none`
see also	`WhiteBalanceValues(), init_color_table()`

6.14 Time Related Functions

c_time	convert system time -> extract time
c_date	convert system time -> extract date
c_timedate	convert system time -> extract date
ltime	convert system time -> extract local time (macro)
ldate	convert system time -> extract local date (macro)
ltimedate	convert system time -> extract local date and time (macro)
gtime	convert system time -> extract GMT time (macro)
gdate	convert system time -> extract GMT date (macro)
gtimedate	convert system time -> extract GMT date and time (macro)
x_timedate	calculate system time
xtimedate	calculate system time and store in system variable SEC (macro)
RTC_set_time	set real-time clock

VC/RT supports a real-time clock with battery backup. On power-up, clock data is loaded into the system variable SEC which represents the number of seconds since 12:00 AM January 1, 1900. The variable SEC and the millisecond counter MSEC are updated by the system when it is running. Time is always stored internally using Greenwich Meantime (GMT). For calculation of local time two system variables (TIMEZONE, DAYLIGHT) are used. So, the first thing to do with a new camera would always be to program the correct timezone and daylight saving time flag. Then check the system time using the time command of the shell. The following functions may be used to convert system time to broken-down time or vice versa. Since the system clock is an interrupt driven process, care should be taken to assure that read-out of the time system variable (system variables) is performed only once for a given set of time variables. Because the time related system variables may change between two accesses, corrupted data may be produced otherwise.

c_time **convert system time -> extract time**

synopsis
```
void c_time(long zsec, int tz, int *sec, int
*min, int *hour)
```

description The function c_time() converts system time passed to the function with the variable zsec into seconds (*sec), minutes (*min), and hours (*hour). The function outputs Greenwich Meantime (GMT) for tz=0 or any other local time for the given timezone (tz).

see also c_date(), c_timedate()

c_date **convert system time -> extract date**

synopsis
```
void c_date(long zsec, int tz, int *day, int
*month, int *year)
```

description The function c_date() converts system time passed to the function with the variable zsec into day (*day), month (*month), and year (*year). The function outputs Greenwich

Meantime (GMT) for tz=0 or any other local time for the given timezone (tz).

see also c_time(), c_timedate()

c_timedate **convert system time -> extract date**

synopsis
```
void c_timedate(long zsec, int tz, int *sec,
int *min, int *hour, int *day, int *month,
int *year)
```

description The function c_timedate() converts system time passed to the function with the variable zsec into seconds (*sec), minutes (*min), hours (*hour), day (*day), month (*month), and year (*year). The function outputs Greenwich Meantime (GMT) for tz=0 or any other local time for the given timezone (tz).

see also c_time(), c_date()

ltime **convert system time -> extract local time (macro)**

synopsis
```
void ltime(int *sec, int *min, int *hour)
```

description The macro ltime() converts system time stored in system variable SEC into seconds (*sec), minutes (*min), and hours (*hour). The function outputs local time with respect to system variables TIMEZONE and DAYLIGHT.

see also ldate(), gdate()

ldate **convert system time -> extract local date (macro)**

synopsis
```
void ldate(int *day, int *month, int *year)
```

description The macro ldate() converts system time stored in system variable SEC into day (*day), month (*month), and year (*year). The function outputs local time with respect to system variables TIMEZONE and DAYLIGHT.

see also ltime(), gtime()

ltimedate **convert system time -> extract local date and time (macro)**

synopsis

```
void ltimedate(int *sec, int *min, int
*hour, int *day, int *month, int *year)
```

description

The macro ltimedate() converts system time stored in system variable SEC into seconds (*sec), minutes (*min), hours (*hour), day (*day), month (*month) and year (*year). The function outputs local time with respect to system variables TIMEZONE and DAYLIGHT.

note:

Be sure to use this function whenever you need a complete set of time and date variables. Using the functions ltime() and ldate() separately might give you an inconsistent set of variables if time changes from 23:59:59 to 00:00:00 of the next day when you call the functions.

see also

ltime(), ldate(), gtimedate()

gtime **convert system time -> extract GMT time (macro)**

synopsis

```
void gtime(int *sec, int *min, int *hour)
```

description

The macro gtime() converts system time stored in system variable SEC into seconds (*sec), minutes (*min), and hours (*hour). The function outputs GMT time.

see also

gdate(), ltime()

gdate **convert system time -> extract GMT date (macro)**

synopsis

```
void gdate (int *day, int *month, int *year)
```

description

The macro gdate() converts system time stored in system variable SEC into day (*day), month (*month), and year (*year). The function outputs GMT time.

see also

ltime(), gtime()

gtimedate **convert system time -> extract GMT date and time (macro)**

synopsis

```
void gtimedate(int *sec, int *min, int
*hour, int *day, int *month, int *year)
```

description

The macro gtimedate() converts system time stored in system variable SEC into seconds (*sec), minutes (*min),

hours (*hour), day (*day), month (*month) and year (*year). The function outputs GMT time.

note: Be sure to use this function whenever you need a complete set of time and date variables. Using the functions gtime() and gdate() separately might give you an inconsistent set of variables if time changes from 23:59:59 to 00:00:00 of the next day when you call the functions.

see also gtime(), gdate(), ltimedate()

x_timedate **calculate system time**

synopsis
```
unsigned long x_timedate(int tz, int sec,
int min, int hour, int day, int month, int
year)
```

description The function x_timedate() converts time and date information into system time which it outputs as return value.

The following parameters are passed to the functions:

tz	timezone	example: 1
sec	second	example: 0
min	minute	example: 59
hour	hour	example: 14
day	day	example: 31
month	month	example: 12
year	year	example: 2001

system time is the number of seconds since 12:00 AM January 1, 1900

see also xtimedate()

xtimedate **calculate system time and store in system variable SEC (macro)**

synopsis
```
void xtimedate(int sec, int min, int hour,
int day, int month, int year)
```

description The macro xtimedate() converts time and date information into system time which it stores in the (long) system variable SEC.

System time is calculated with respect to system variables TIMEZONE and DAYLIGHT.

parameters The following parameters are passed to the functions:

sec	second	example: 0
min	minute	example: 59
hour	hour	example: 14
day	day	example: 31
month	month	example: 12
year	year	example: 2001

system time is the number of seconds since 12:00 AM January 1, 1900

see also x_timedate()

RTC_set_time **set Real Time Clock**

synopsis `void RTC_set_time()`

description Programs Real Time Clock Chip according to Systems variables set by xtimedate

example : time command of the shell

```
time_sopt()
{
  int sec,minute,hour,day,month,year;
  display_timezone();
  ltimedate(&sec,&minute,&hour,&day,&month,&year);
  print("time: %02d:%02d:%02d\n",hour,minute,sec);
  print("date: %02d/%02d/%02d\n",month,day,year-2000);
  enter_timezone();
  enter_date(&day,&month,&year);
  enter_time(&hour,&minute,&sec);
  xtimedate(sec,minute,hour,day,month,year+2000);  //set
internal clock
  setvar(LOWBAT,0); /* reset internal lowbat */
  RTC_set_time(); /* program clock chip */
}
```

see also xtimedate()

TIMER2 Macros

For the VC40xx and VC44xx Smart Cameras, there is a user programmable interrupt timer available, TIMER2. TIMER2 may be programmed using macros. The zero-interrupt is available as an event.

The following macros are available:

`TIMER2_RESET()`	resets TIMER2 to its default state
`TIMER2_INIT(T2CTRL,nclk,0)`	initializes TIMER2 to `nclk` = *number of clocks*
`TIMER2_START()`	starts TIMER2
`TIMER2_STOP()`	stops TIMER2

Whenever TIMER2 counts down to zero, an event (`TIMER2`) is generated. See the chapter about events for further information.

For the VC6XXX and VC7XXX cameras contact VC support for a special demo of TIMER2 macros.

7 Prototypes, Include Files

The file <vcrt.h> contains the corresponding prototypes for all functions described in this documentation.

It is especially important to add this include file to your user program if you call functions with variable argument lists (print(), exec()).

This is usually done by adding the command

```
#include <vcrt.h>
```

to the beginning of the C program file.

The file `<register.h>` contains hardware dependent declarations, the file `<sysvar.h>` the declaration of the system variables. (See discussion of the system variables in Appendix E).

You may also wish to include the header file `<vlib.h>` which is part of the VCLIB image processing library package not covered here.

8 Memory Model of VC20xx / VC4xxx / VC6xxx Cameras

In contrast to the ADSP2181 signal processor, the TMS320C62xx used in the VC20xx cameras and the TMS320C64xx used in the VC40xx and VC44xx cameras has only one unified memory space. There are 16, 32, 64, 128 and 256 MByte versions available for the DRAM memory.

The DRAM memory used is organized in 4 pages of equal size. The DSP is able to keep all 4 pages open at the same time. If used properly this feature me be used to speed up programs.

The following table summarizes some information about the memory:

memory size	**16 Mbytes VC20XX SDR**	**32 Mbytes VC40XX SDR**	**64 Mbytes VC44XX SDR**	**128 Mbytes VC62XX DDR2**	**256 Mbytes VC64XX DDR2**
start address	0xA0000000	0xA0000000	0xA0000000	0x80000000	0xE0000000
end address	0xA0FFFFFF	0xA1FFFFFF	0xA3FFFFFF	0x87FFFFFF	0xEFFFFFFF
size (hex)	0x01000000	0x02000000	0x04000000	0x08000000	0x10000000

9 Functional Principle of the VC20xx / VC40xx / VC44xx / VC62XX / VC64XX Smart Cameras

Figure 1 illustrates how the cameras work. The differences between the various camera types have to do with the CCD sensors used and the frame output, for which different extension boards are used.

The left side of the figure shows the sensor board, with the CCD sensor, the controller and processing of the video signal.

The controller is used to read-out the CCD sensor, like for common cameras. The controller's modes can all be set by software.

The output of the CCD sensor is an analog signal, which is passed to a programmable gain amplifier (PGA, software programmable) and then to the A/D converter. The A/D conversion used is called "pixel-identical", because there is a separate gray value for each pixel of the CCD sensor.

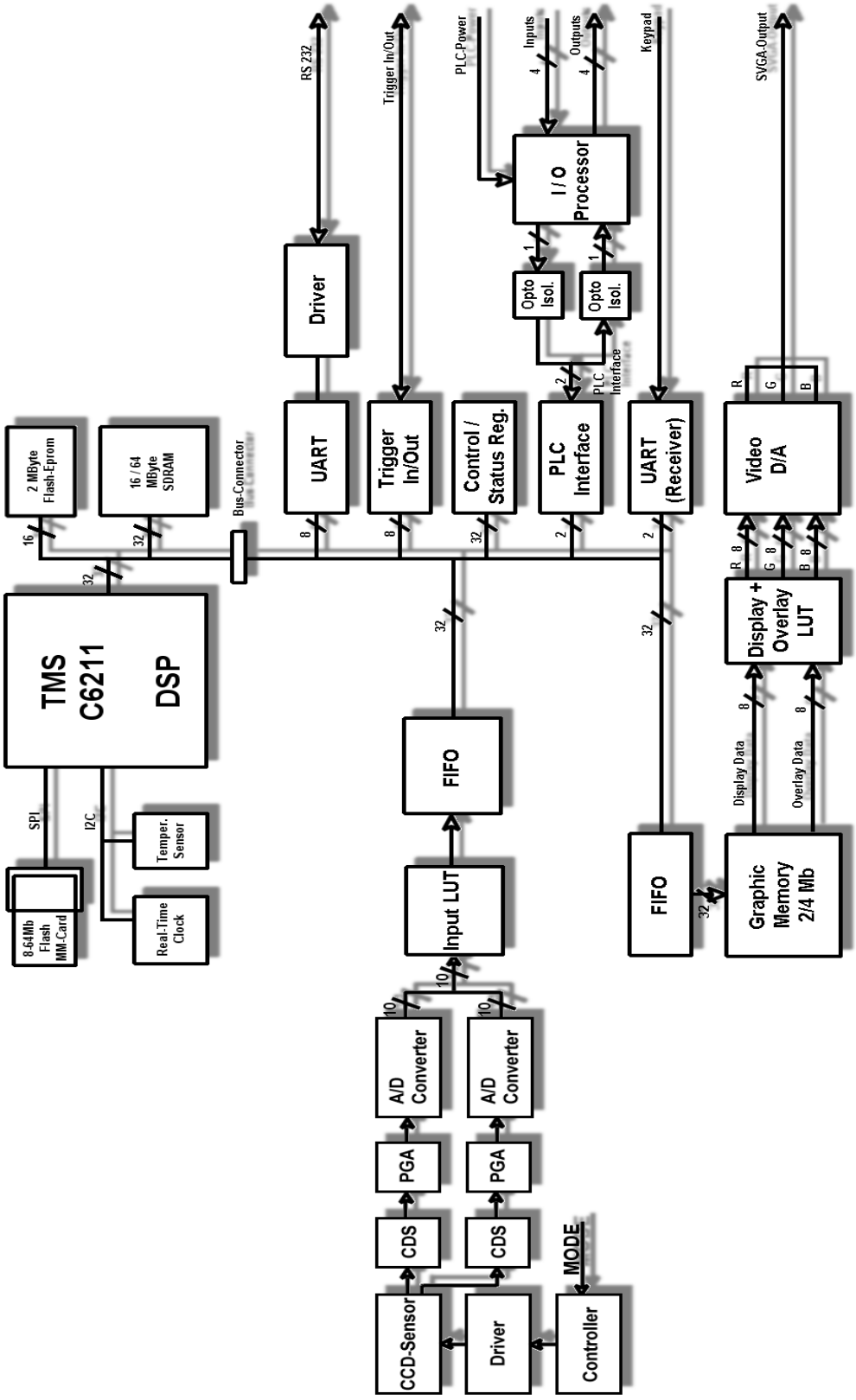
The video data may be modified using an input LUT. The image information is then stored in the DSP's main SDRAM memory using DMA.

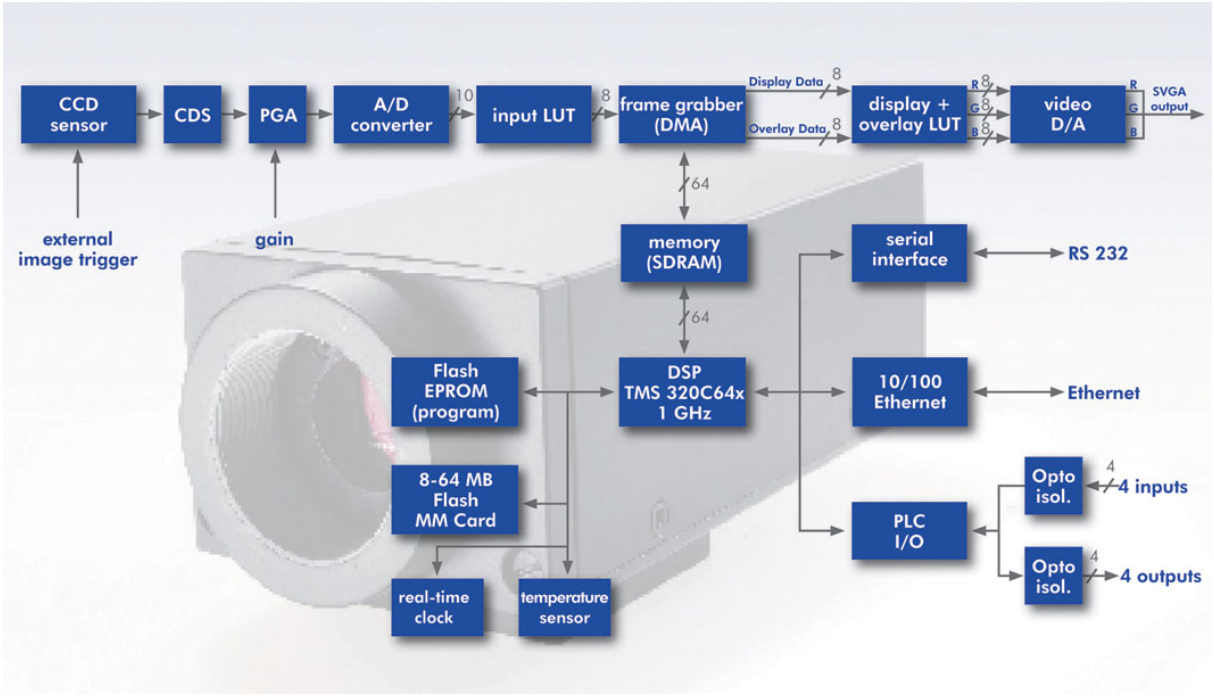
The image may then be displayed on the monitor in real time or as a stored image. Therefore, part of the main memory is copied to the "Graphic Memory" via DMA. This data transfer is usually active continously guaranteeing that the monitor will always display up-to-date information. The image displayed on the screen first passes a color LUT and is then displayed as 24bit RGB graphics. It may be combined with overlay data which is also displayed in 24bit color using a second LUT.
For VC40xx and VC44xx Smart Cameras the video display is done directly from the main SDRAM memory; no "Graphic Memory" is needed.

For external control of the image acquisition process a fast trigger input is provided. A trigger output may be used to trigger a strobe light. Both functions are fully implemented in hardware.

Taking and reproducing pictures is almost 100% supported by hardware. This means, it does not require computing time. It does, however, consume memory bandwith. It is quite difficult to tell if this will slow down processing and how much. To be on the safe side, it is recommended to avoid these functions wherever it is possible. (e.g. displaying a stored image is better than a live display). As a ballpark number, the image ackquisition may delay program execution by perhaps 1%.

9.1 Block Diagram of VC20xx Cameras


RS 232
Trigger In/Out
PLC-Power
Inputs
Outputs
Keypad
SVGA-Output
I / O Processor
Driver
Opto Isol.
Opto Isol.
PLC Interface
2 MByte Flash-Eprom
16 / 64 MByte SDRAM
Bus-Connector
UART
Trigger In/Out
Control / Status Reg.
PLC Interface
UART (Receiver)
Video D/A
TMS C6211 DSP
FIFO
Display + Overlay LUT
SPI
I2C
Temper. Sensor
8-64Mb Flash MM-Card
Real-Time Clock
Input LUT
FIFO
Graphic Memory 2/4 Mb
Display Data
Overlay Data
A/D Converter
A/D Converter
PGA
PGA
CDS
CDS
MODE
CCD-Sensor
Driver
Controller



9.2 Blockdiagram VC44xx

9.3 Blockdiagram VC401X aka SBC4000

9.4 Blockdiagram VC60XX / VC62XX

9.5 Blockdiagram VC64XX

10 Organization of the DRAM

The VC20xx / VC40xx / VC44xx series cameras are equipped with SDRAM (synchronous dynamic RAM) for storage of large amounts of data. The size of this SDRAM memory ranges from 16 MBytes for the VC20xx cameras to more than 128 MBytes for the VC44xx cameras. VC20xx and VC40xx cameras have a 32Bit wide organization of the memory, VC44xx cameras have 64Bit organization. The SDRAM is used for main memory, program, data and video data (images). It is volatile, meaning the data is lost when the supply voltage is switched off. Smart cameras of type VC4016/18 do not have a video output.

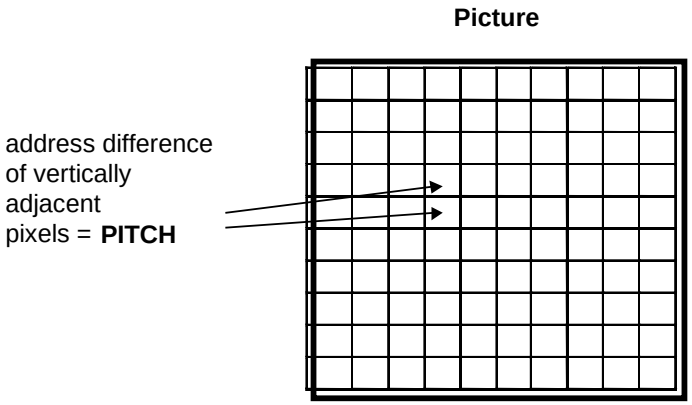
Organization of the video memory:

> **Note**, that the mapping of pixels to bytes has changed with respect to prior versions with ADSP2181 DSP. (VC20xx / VC4xxx / VC6xxx / VC7XXX cameras use little endian byte mapping).

The video memory can be any part of the SDRAM. The size of this memory area depends on the frame format and the number of required frames. A start address can be specified individually for the SDRAM position of the picture taken or shown on the screen (system variables `CAPT_START`). or `DISP_START`). This makes it possible to display several video memory screens, for example, or to take several pictures in rapid sequence. They can then be processed, etc.
The system automatically allocates memory for one image (`size = (DHWIDTH, DVWIDTH)`) and sets `CAPT_START` and `DISP_START` to the same address, so that all the ackquired images will be displayed automatically.

Based on the start address, the picture is written to the subsequent memory area or read from it. The first pixel (for addr=startad) is located in the upper left corner of the picture. The next pixel is directly to its right in the same line, etc. This way, an entire line is stored in a continuous memory area.

To get to the beginning of the next line, the value "pitch" must be added to the beginning of the previous line (in this case, startad). The correct value for pitch depends on how the picture format was programmed, thus on the camera type.



The picture format used may results in some unused memory. For example, if the pitch were 1024 and the number of pixels per line 744, this results in 1024-744=280 bytes (about 30%) which are wasted per line. The memory space could be utilized better either by reducing the number of pixels per line (e.g. cols=512, pitch=512) or by copying the picture to a compact memory area.

active area of the video memory 744 columns 574 lines	unused area 1024-744=280 columns

11 Organization of the Overlay DRAM

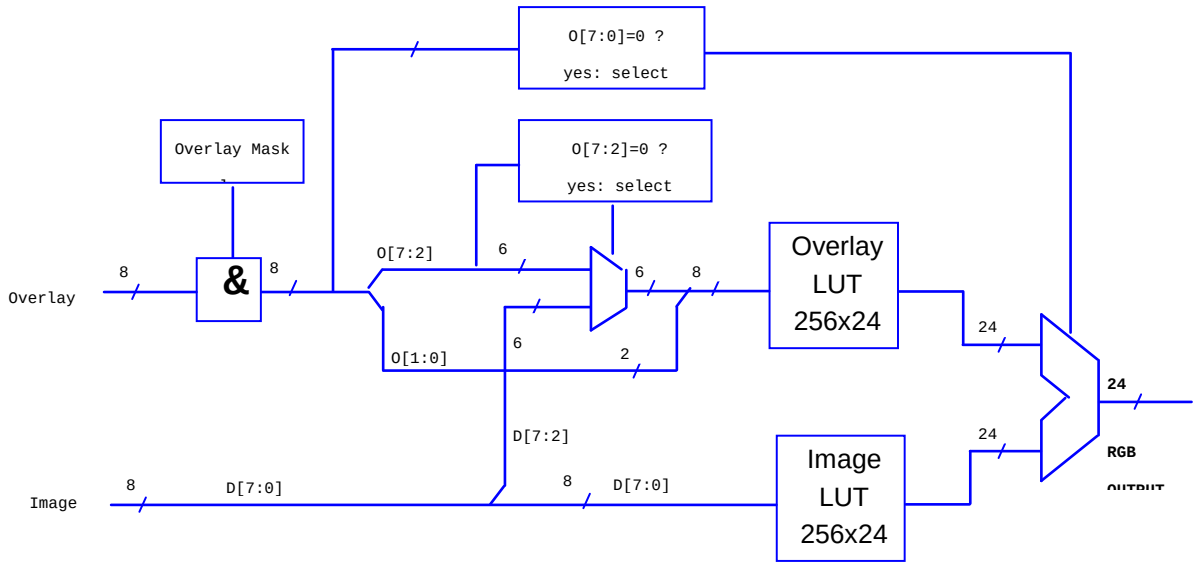
Just like the video memory, the overlay memory can be any part of the SDRAM. You must of course make sure that the overlay memory does not overlap video memory or data memory areas. A start address can be specified for the overlay. The system variable OVLY_START in the header file sysvar.h is used for this.

The organisation of the overlay SDRAM is the same as for the video data SDRAM. Like the latter, 8 bits per pixel are used. If the pixel's value is zero, the overlay is inactive and video data will be displayed. If the pixel's value is nonzero, overlay information will be displayed depending on the state of the overlay mask register.

With the exception of camera models without video output (e.g. VC401X / VC6XXX), the VC20xx / VC40xx / VC44xx cameras feature powerful image graphics and overlay display features.

- 8 bit image graphics plus independent 8 bit overlay
- 2 lookup tables 256x24 (RGB) for image and overlay
- 2x3 lookup tables for color cameras
- 8 bit overlay mask for individual control of overlay bits
- 6 regular overlay planes + 3 translucent overlay planes

The following drawing gives an overview of the functionality:



It is important to know that there is a memory for image data starting at address DISP_START in maín memory. This data is normally displayed using the "Image LUT" . Besides that the user may use an overlay memory with the same size (and organized with 8 bits per pixel) starting at address OVLY_START in main memory. Depending on the bits set in overlay memory and the value of the overlay mask the pixel will be displayed either as overlay using the "Overlay LUT" , as image using the "Image LUT" or as a combination of both (6 bits from the image and 2 bits from overlay) using one of the three translucent tables in the "Overlay LUT".

With the pixel mask register it is possible to select and deselect individual overlay planes very rapidly. Setting the register to zero disables the overlay display.

The following table summarizes the functionality of the image data and overlay display:

O[7..0] = 0	no overlay, display of image data through image data LUT
O[7..2] ≠ 0	normal overlay, display of overlay data through overlay LUT
O[7..2] = 0, O[1..0] ≠ 0	3 translucent tables, display of image data through overlay LUT

12 Description of the File Structure

Start address of the file system is at address 0x080000 (sector 8). User files can be stored starting at address 0x100000 (sector 16). The files are stored one after another, without gaps.

Here's the overview about the different file types :

- Executable File
- ASCII File
- Binary Data File
- JPEG Data File
- RLC Data File

FLASH EPROM FILE STRUCTURE

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	See table below
File name	3	9 bytes	in ASCII code with \0 as end, i.e. a maximum of 8 characters plus \0
Number of modules	12	2 bytes	Always 0001 = 1 module
Dummy	14	2 bytes	reserved for later use
Module type	16	1 byte	00
Length	17	4 bytes	length
Data	21	n bytes	n=length
Check sum		1 byte	currently 0x55

File types

File type	File extension	Hex code
Executable file	`exe, out`	`0x00`
ASCII	`asc, txt, htm`	`0x01`
BINARY	`dat`	`0x02`
JPEG	`jpg`	`0x03`
RLC	`rlc`	`0x04`
compressed executable	`cex`	`0x80`
compressed ASCII	`cas`	`0x81`
compressed BINARY	`cda`	`0x82`
compressed JPEG	`cjp`	`0x83`
compressed RLC	`crl`	`0x84`

The internal data structure for executable files complies to the standard .COFF format.

13 System Variables

VC/RT allows access to a series of system variables. Their addresses are defined in a header file called `sysvar.h`. Please always use the names in this header file as a reference. Do not use absolute addresses, as they may be changed while the development of the cameras continues. System variables may be accessed using the functions `getvar()` , `setvar()` , `getlvar()` and `setlvar()`.

The following is a list of the most important system variables:

Variable	mode	description
DISP_PERIOD	r/w	refresh rate for display & overlay
DISP_CNT	r/w	counter for refresh rate (counts down)
DISP_START	r/w	start address for display (must be multiple of 1024)
OVLY_START	r/w	start address for overlay (must be multiple of 1024)
DISP_ACTIVE	r/w	0: no refresh / 1: refresh (display)
OVLY_ACTIVE	r/w	0: no refresh / 1: refresh (overlay)
CAPT_START	r/w	start address for image capture (must be multiple of 1024)
HWIDTH	r/o	sensor active horizontal pixels
VWIDTH	r/o	number of active vertical sensor lines
VPITCH	r/o	video pitch
EXPCNT	r/w	number of exposure cycles (lines)
GAIN	r/w	video gain value
IMODE	r/w 1.)	video mode, 0=life refresh, 1=stop after current image
VSTAT	r/w 1.)	video status 0=idle 1=capture busy
CPUCLK	r/o	master cpu clock frequency
MSEC	r/w 2.)	real-time clock: millisecond
SEC	r/w 2.)	real-time clock: seconds since 1900 (long value)
EXUNIT	r/w	time unit for exposure control [usec]
TIMESTAMP	r/o	timestamp for last captured images [ms]
DAYLIGHT	r/w	daylight saving time flag
TIMEZONE	r/w	real-time clock: timezone
LOWBAT	r/o	low battery voltage: 1=time invalid 0=time ok
TEMP	r/o	cpu board temperature
VERSION	r/o	VCRT software version
DRAMSIZE	r/o	size of main SDRAM
PLCOUT	r/w	state of the PLC outputs
PLCIN	r/o	state of the PLC inputs
POWFAIL	r/o	1: PLC power failure / 0: power ok
EXPOSING	r/o	tracking number of the image being *exposed*
STORING	r/o	tracking number of the image being *stored*
IMGREADY	r/o	tracking number of the last image being *ready* for processing
LATENCY	r/w	maximum interrupt latency (testversions only)
MMC	r/o	missing multi-media / sd card: -1
IPADDR	r/o	IP address (ethernet version)
IPMASK	r/o	IP mask (ethernet version)
IPGATE	r/o	IP gateway (ethernet version)
DHCP	r/o	dhcp 1=on 0=off -1=failure
TPRIORITY	r/w	exec2( ) task priority default=9
FPGAVERSION	r/o	fpga version / date
OVC_STAT	r/o	overcurrent status (DM640 only)
SCRLOGPAGE	r/w	needed for macros.h
OVLLOGPAGE	r/w	needed for macros.h
MODEL	r/o	camera model
DHWIDTH	r/o	display horizontal width
DVWIDTH	r/o	display vertical width
OVL_MASK	r/w	overlay mask default value
PRIVATE	r/o	index for private sysvars
TELNET	r/o	telnet active
TOTAL	r/o	total number of clocks in line
XSG	r/o	clock cycles between XSUB & XSG
USR_EVENT	r/o	first user event number

USR_EVT_LAST	r/o	last user event number
RED	r/w 3.)	whitebalance RED value
GREEN	r/w 3.)	whitebalance GREEN value
BLUE	r/w 3.)	whitebalance BLUE value
GAMMA	r/w 3.)	gamma for output LUT
RGBO_START	r/w	start of RGBO buffer / color camera
COLOR_MODE	r/w 3.)	mode for color display
UPTIME	r/w 2.)	system uptime in seconds
SYSMEM	r/o	system mem struct
TESTVERSION	r/o	testversion 0 = release
ETHLINK	r/o	ethernet link info (1 = link active, 0 = no link)
LTEST		interrupt latency test
EMAC_COUNT		emac_count
TIME_SLICE	r/w	time_slice
SENSORID	r/o	sensor id of camera head
DEFHANDLE	r/o	nitial io default handle
DEBUG_SWITCH	r/w	switch to enable debugging info
DNSSERVER1	r/o	dns server 1 ip address
DNSSERVER2	r/o	dns server 2 ip address
EXUNIT_F	r/o	time for exposure control [us] float
GAINDIFF	r/w	offset for second ADC values +/-32
PPW	r/o	password pointer
SDCARD	r/w	sdcard options
CAPT_START2	r/w	start address second image capture
MONI_TICKS	r/w	polling ticks of monitor task in ms
IOPORTIN		VC64XX IO PORT INPUT value
IOPORTOUT		VC64XX IO PORT OUTPUT value
IOPORTDIR		VC64XX IO PORT DIR value
SENSORID2	r/o	sensor id of camera head 2
DHCPSERVER	r/o	dhcpserver
VC_HW_TYPE	r/o	VC Hardware platform type
STEREO_MODE	r/w	VC64XX stereo mode 0=compat 1=offset
PRIVATESYS	r/o	storage for 50 private sysvars

r/w = read / write
r/o = read / only (it is not allowed to write to this system variable)
1.) changed by image capture process
2.) changed by timer tick
3.) changed by shell command

Please note, that most of the system variables are **highly hardware dependent**, e.g. the variables HWIDTH and VWIDTH reflect the size of the active sensor area in horizontal and vertical direction.

In the following some of the system variables are explained in detail:

DISP_PERIOD is the refresh rate for display & overlay in units of display cycles. It is only applicable to model VC20xx cameras. (Model VC40xx refresh the display directly from main memory) DISP_PERIOD is always 1 regardless of the value written into this register). E.g. with a display refresh rate of 70 Hz, one display cycle would be 14 milliseconds. A value of 4 (default) for DISP_PERIOD means that the video refresh memory would be updated from main memory each 4*14 = 56 milliseconds.

DISP_CNT is a counter counting down from the value written to DISP_PERIOD to 1. Whenever it reaches 0, it is automatically reloaded to DISP_PERIOD and the video refresh takes place.

DISP_START, OVLY_START, CAPT_START store the address of the memory buffers for display, overlay and capture. The system stores default values for allocated memory on system start. The default capture and display address are equal, i.e. whenever an image is captured, it will be displayed on the video monitor. Since smart cameras like the VC4018 and VC4016 do not provide display overlay, OVLY_START is zero for those cameras. If the user needs the overlay memory for compatibility reasons, it is possible to allocate the proper overlay space and write the start address to OVLY_START.

DISP_ACTIVE and OVLY_ACTIVE allow enabelling and disabelling the refresh of the display and overlay buffers separately. This feature is available for VC20xx smart cameras only.

HWIDTH and VWIDTH are the horizontal and vertical size of the **sensor** in pixels.
DHWIDTH and DVWIDTH are the horizontal and vertical size of the **display** in pixels. For some camera models the display size is larger than the sensor size, for others both sizes are equal. For smart cameras without video output, e.g. the models VC401X and VC6XXX **the values of DHWIDTH and DVWIDTH are zero** !

VPITCH is the video pitch, i.e. the address difference of two vertical neighbor pixels. There is only one video pitch, i.e. the pitch for capture and display is the same.

IMODE and VSTAT are set and used by the image capture routines like tpict() and vmode(). IMODE=0 indicates a live mode image refresh, i.e. the system captures images at the fastest rate possible. VSTAT=1 signals that a capture is currently active. It should be noted, that the function capture_request() does not use these variables, they are used by the functions tpict(), tenable(), tpstart(), tpwait(), trdy() and tpp() only.

MSEC and SEC: like other system variables these values can change on the fly. So please make sure that the values for MSEC and SEC are consistent, when reading both.

TIMESTAMP is a pointer to a struct where timestamp information for a series of images is stored. See the chapter about image capture timestamp operation for further information.

TEMP is the CPU board temperature. The value stored in this variable is twice the temperature in degrees Celsius, i.e. it has a resolution of 0.5 degrees.

POWFAIL is available for all cameras with separate PLC power supply, namely the VC20xx and the VC40xx cameras excluding the VC401X and the VC6XXX.

EXPOSING, STORING and IMGREADY reflect the status of the image capture queue. The tracking number of each image (which is the return value of the function capture_request()) is automatically written to these variables according to its state.

IPADDR, IPMASK, IPGATE are 32bit (Hex) values for IP address, mask and gateway. They are applicable for Ethernet cameras only and cannot be changed by the user. Changing the IP address requires changing the system file #IP.txt on the device fd: and performing a power-up sequence.

TPRIORITY and TIME_SLICE are used when calling the function exec2() for starting a parallel process. Higher values indicate a lower priority for TPRIORITY. For processes with equal priority it is possible to work with timeslices. Simply write the timeslice value in milliseconds to the system variable TIME_SLICE.

OVC_STAT is used for VC4018 and VC4016 smart cameras. If its value is zero, the PLC outputs work normally. When there is an overcurrent situation (i.e. the current flowing through all PLC output terminals exceeds a threshold like 1 or 2 amps), all the PLC outputs are switched off, OVC_STAT is set to a system dependend start value, which counts down. When this value reaches zero, the system switches the outputs to their former state in order to test the overcurrent condition and to return to normal operation when the short-circuit has disappeared.

SCRLOGPAGE and OVLLOGPAGE: it is possible to use physical and logical pages for image and overlay display.

OVL_MASK is a copy of the hardware overlay mask used for overlay video display. It is set and updated by the function set_ovlmask().

PRIVATE: the value of this system variable indicates at which system variable number an array of 50 user defined system variables begin.

TELNET: this system variable is 1 when a telnet connection is open, otherwise its 0.

The value of USR_EVENT indicates the first event number available to the user as a user event. USR_EVENT_LAST is the last available user event number.

RED, GREEN, BLUE are the whitebalance values for color cameras with hardware whitebalance support. The function **WhiteBalanceValues()** is used to calculate the values for the red, green and blue channels. A value of 1024 for a channel means, that the channel is used one-to-one, i.e. without any change. A value larger than 1024 corresponds to an amplification of that channel, e.g. 2048 would be an amplification by a factor of 2. There is always at least one channel with a value of 1024. The function **init_color_lut()** is used to program the hardware lookup-table for the three channels. This function also sets the values for the system variables RED, GREEN and BLUE for further reference. A whitebalance can also be done using the shell command *wb*.

GAMMA is used to compensate display monitor non-linearities. The value of the system variable GAMMA is divided by 100 and used as an argument for the function **init_LUT_gamma()**. This function then programs the output lookup-tables in the appropriate way. See the documentation of the function **init_LUT_gamma()** for further information. The lookup-table and the value of GAMMA can be changed using the shell function *disp –g*.

COLOR_MODE is applicable only for color cameras with video output. It is used to specify the video output mode according to the following table:

0	IDLE	no display, maximum CPU performance
1	GREY	display of a black-and-white (grey) image
2	RGB	display of an image in RGB format
3	BAYER	display of an image in Bayer-pattern format in full color
4	BAYERGREY	display of an image in Bayer-pattern format as black-and-white image
5	YCBCR	display of an image in YCbCr format

Changing the system variable COLOR_MODE instantly changes the mode of the display. This can also be done using the shell command *disp –c*.

UPTIME is the time in seconds since the start of the system (hardware boot or software re-boot).

ETHLINK is the Ethernet link information. A value of 1 means that the system has detected an Ethernet PHY on the remote computer site and a link is present. Otherwise the value is 0.

Example: How to use Systems Variables

```
#include <sysvar.h>

void set_display_start(int addr)
{
  setvar(DISP_START, addr);   /* Use of system variable
DISP_START */
}
```

14 Image Capture Timestamps

Whenever an image is captured, a timestamp for this image is stored in a table together with its tracking number for further reference. The system variable `TIMESTAMP` provides the pointer to this table. The number of elements in this table is given by `IMGTS_SIZE` which is currently set to 20.

The table has the following format:

```
typedef struct
{
  long long exptimestamp;     /* time stamp of image  */
  int       imageno;          /* image number         */
} imgts;
```

The timestamp value is calculated according to the following formula:

```
exptimestamp  = 1000 * getvar(SEC) + getvar(MSEC)
```

The following progam may help to understand the timestamp feature:

```
print("exposure timestamps  : 0x%08lx\n",getvar(TIMESTAMP));
  {
  int i;
  imgts * ts_table = (imgts *)getvar(TIMESTAMP);

  for(i=1;i<=IMGTS_SIZE && ts_table;i++,ts_table++)
    {
    print("%02d (0x%x) ",i,ts_table);
    print("nr= %d ",ts_table->imageno);
    print("ts= %lu\n",ts_table->exptimestamp);
    }
  }
```


15 Useful Files

The following batch files (.BAT files) are useful for working with the development system. After VC/RT is installed, these files are located in the corresponding VC/RT directories.

15.1 c.bat

```
cl6x -o3 –mi100000 -ml3 –pl %1.c
```

This batch file is used to compile a program without calling the linker.

It is usually used for large projects. Each C source file can be compiled individually and then linked with another batch file.

Call:

```
c pgm1
```

This call compiles the program pgm1.c and creates the object file pgm1.obj.

The option

```
-o3
```

compiles for the best optimization possible.

```
-mi100000
```

specifies a threshold of 100000 cycles for blocking the system interrupts. Without this option the compiler may block the system interrupts for an extended period of time which may result in serious system failures

```
-ml3
```

compiles for the "large" memory model. Without this option, the program is further optimized.

15.2 cc.bat

```
cl6x -o3 –mi100000 -ml3 -pl %1.c
lnk6x –s -u _c_int01 %1.obj -m %1.map -o %1.out cc.cmd
strip6x %1.out

copy %1.out exec.out
\adsp\21xx\util\econv %1
\adsp\21xx\util\scvt
copy adsp.msf %1.msf
```

This batch file compiles and links a program, and converts it to S Records. The .msf file thus created is then copied to the current directory. The .msf may then be downloaded to the camera using the lo- command. Alternatively, the .out file could be transferred to the camera via FTP.

This batch file compiles only a single C source file. If the program consists of several source files, they can be individually compiled and linked with, say, C.BAT.

Call:

```
cc pgm1
```

This call compiles the program pgm1.c. It creates the files pgm1.out and pgm1.msf in the working directory

cc.bat links your program with the Texas Instruments runtime library and the Vision Components libraries vcrt.a and vclib.a.

The `-s` option of the linker and the command `strip6x` remove all unnecessary information in the output file. For debugging purposes, it may be helpful to have this information. In this case remove both from the batch file.

This batch file also produces a loader map pgm1.map .

15.3 cc.cmd

The linking process is controlled by the file cc.cmd

```
-c
/* -priority */ /* CCS 3.0 and above */
-l vcrt4.lib
-l vclib.lib
-l extlib.lib
-l colorlib.lib
-l flib.lib
-l rts6200.lib
-u _c_int01
-e _c_int01
-stack 0x4000 /* adjust appropriate - stack size: min=0x4000 max depends on camera max mem */
-heap  0x400  /* adjust appropriate - heap size : min=0x400  max depends on camera max mem */

MEMORY
{
    PMEM:   o = 0a0200000h       l = 100000h /* intended for initialization   */
    BMEM:   o = 0a0090000h       l = 40000h  /* .bss, .system, .stack, .cinit */
}

SECTIONS
{
    .text       >       PMEM
    .tables     >       PMEM
    .data       >       PMEM
    .stack      >       BMEM
    .bss        >       PMEM
    .sysmem     >       PMEM
    .cinit      >       PMEM
    .const      >       PMEM
    .cio        >       PMEM
    .far        >       PMEM
}
```

Here, the libraries are specified (`vcrt4.lib`, `vclib.lib`, `extlib.lib`,`colorlib.lib`, `flib.lib` , `rts6200.lib`)
The stack size (`-stack 0x4000`), the heap size (`-heap 0x400`), and the memory map are specified. The stack size is only valid if the program is loaded as a parallel task into the module directory. The heap size is important if the function uses the TI-function `malloc()`. This may be the case for most of the C++ programs, where it is recommended to specify a large heap space.

15.4 Large Projects

For large projects consisting of several C source files, it is easy to create your own .BAT files for compiling and linking.

The following illustrates how to do this, based on the .BAT files used when creating the operating system.

The individual C files can be compiled with, say, C.BAT.

To compile all C files, a .BAT file called MAKE.BAT can be used. Of course, this file must be tailored to each project.

Please do not forget to change this file whenever you add or delete C files from the project.

```
cl6x -o3 -ml3 loader.c
cl6x -o3 -ml3 rs232.c
cl6x -o3 -ml3 rs232a.c
cl6x -o3 -ml3 setbaud.c
cl6x -o3 -ml3 fnaddr.c
cl6x -o3 -ml3 search.c
cl6x -o3 -ml3 coldport.c
cl6x -o3 -ml3 main.c
cl6x -o3 -ml3 bd.c
cl6x -o3 -ml3 del.c
cl6x -o3 -ml3 dir.c
cl6x -o3 -ml3 dwn.c
cl6x -o3 -ml3 dmp.c
cl6x -o3 -ml3 dd.c
cl6x -o3 -ml3 er.c
cl6x -o3 -ml3 ex.c
cl6x -o3 -ml3 fd.c
cl6x -o3 -ml3 go.c
cl6x -o3 -ml3 he.c
cl6x -o3 -ml3 ht.c

lnk6x –s -u _c_int01 shell.obj -m shell.map -o shell.out shell.cmd
strip6x shell.out
copy shell.out exec.out
\adsp\21xx\util\econv shell
\adsp\21xx\util\scvt
copy adsp.msf shell.msf
```

Our MAKE.BAT contains a linker call, but we usually use a second batch file (L2.BAT) for linking and creating the .MSF file.

```
lnk6x -u _c_int01 shell.obj -m shell.map -o shell.out shell.cmd
strip6x shell.out
copy shell.out exec.out
\adsp\21xx\util\econv shell
\adsp\21xx\util\scvt
copy adsp.msf shell.msf
```

This calls the linker (lnk6x) with a reference to the file shell.cmd. This option causes the linker to read the file names required for linking the project from the file shell.cmd.

For our project, shell.cmd must contain the following:

```
loader.obj
rs232.obj
rs232a.obj
setbaud.obj
fnaddr.obj
search.obj
coldport.obj
main.obj
bd.obj
del.obj
dir.obj
dwn.obj
dmp.obj
dd.obj
er.obj
ex.obj
fd.obj
go.obj
he.obj
ht.obj
```

This file must be modified as the project develops. All objects not listed here are taken from either the run-time library rts6200.lib (rts6400.lib) or from the VCRT library.

15.5 Relocatable Objects

The linker allows to create relocatable objects. This is necessary if parallel processes need to be started using the relocatable loader of the VCRT operating system. The relocatable loader loads the programs not to the addresses for which they originally have been linked, but to memory addresses where the system allocates memory for this program. This method is thus very flexible and convenient. The load addresses of the programs may be listed using the mdir shell command.

Relocateable objects may be created using the batch file:

ccr.bat

```
cl6x -o3 -mi100000 -pl %1.c
lnk6x -ar -u _c_int01 %1.obj -m %1.map -o %1.out ccr.cmd
strip6x %1.out

copy %1.out exec.out
..\util\econv %1
..\util\scvt
copy adsp.msf %1.msf
```

ccr.cmd

```
-c
/* -priority */ /* CCS 3.0 and above */
-l vcrt4.lib
-l vclib.lib
-l extlib.lib
-l colorlib.lib
-l flib.lib
-l rts6200.lib
-u _c_int01
-e _c_int01
-stack 0x4000 /* adjust appropriate - stack size: min=0x4000 max depends on camera max mem */
-heap  0x400  /* adjust appropriate - heap size : min=0x400  max depends on camera max mem */
```

```
MEMORY
{
    PMEM:   o = 0h       l = 0ffffffffh
}

SECTIONS
{
.text   : ALIGN(32) { *(.text) } > PMEM
.const  : ALIGN(8) {} > PMEM
.data   : ALIGN(8) {} > PMEM
.bss    : ALIGN(8) { *(.bss)   } > PMEM
.cinit  : ALIGN(4) { *(.cinit) } > PMEM /* cflag option only */
.pinit  : ALIGN(4) {} > PMEM /* cflag option only */
.stack  : ALIGN(8) {} > PMEM /* cflag option only */
.far    : ALIGN(8) {} > PMEM /* cflag option only */
.sysmem: ALIGN(8) {} > PMEM /* cflag option only */
.switch: ALIGN(4) {} > PMEM /* cflag option only */
.cio    : ALIGN(4) {} > PMEM /* cflag option only */
}
```

16 Description of the Example Programs

16.1 test.c

This is the first program you should compile to check if everything works correctly.
The program just outputs:

```
hello world !!!!
```

16.2 info.c

The program "info" outputs a series of system variables via the serial interface. For example, the image format can be determined. The following is a copy of the program's printout running on a VC51:

```
$info

**********************
* System-Variables   *
**********************

cpu clock frequency  : 39321600
current video line   : 39
startpage of image   : 0
startaddress image   : 0x0
active hor. pixels/2 : 372
active ver. pixels   : 574
pitch / 2            : 512
startpage overlay    : 143
startaddress overlay
byte address         : 0x00047700
bit address          : 0x0023B800
overlay pitch / 16   : 64
Offset_Overlay       : 2048
overlay hw offset    : 46

$
```

17 List of VC/RT Functions

Memory Allocation Functions

Name	Type	Description
`void prtfree(void)`	M	Print available memory segments
`void *vcmalloc(unsigned int size)`	M	Allocate memory
`void vcfree(void *ptr)`	M	Release memory
`void *sysmalloc (unsigned nwords, int type)`	S	Allocate system memory
`void sysfree (void *ap)`	S	Release system memory
`void sysprtfree (void)`	S	Print available system memory segm.
`U8 *DRAMScreenMalloc(void)`	M	allocate DRAM for full screen storage

General I/O Functions

Name	Type	Description
`FILE *io_fopen(char *path, char *mode)`	C	open a device, get file pointer
`int io_fclose(FILE *fp)`	C	close a device
`int io_read(FILE *fp, char *buf, int cnt)`	C	read from device
`int io_write(FILE *fp, char *buf, int cnt)`	C	write to device
`int io_ioctl(FILE *fp, unsigned cmd, void *param)`	C	I/O control
`int io_fgetc(FILE *fp)`	C	get character from device
`int io_fputc(int c, FILE *fp)`	C	output character to device
`int io_fseek(FILE *fp, int offset, unsigned start_from)`	C	set the file position
`FILE *io_get_handle(unsigned stdio_type)`	C	get a pointer to the default standard I/O stream
`I32 *io_pipe_install(char *name, U32 size)`	C	install a pipe device

Program Execution

Name	Type	Description
`int exec(char *fname, p1,p2, ... , pn)`	S	Load and execute a program
`int exec2(char *fname, p1,p2, ... , pn)`	S	Load and execute a program as a parallel task

I/O Functions

Name	Type	Description
`void pstr(char *str)`	C	Output a string via the serial interface
`void print(char *format, ...)`	C	Formatted output of text and variables
`void sprint(char *s, char *format, ...)`	C	Formatted output of text and variables to a string
`int hextoi(char *s)`	C	convert hex value string to integer
`void setRTS(void)`	M	set RTS signal
`void resRTS(void)`	M	reset RTS signal
`void setPLCn(void)`	M	set PLC signal
`void resPLCn(void)`	M	reset PLC signal
`void outPLC(int value)`	S	output value to PLC
`int inPLC(void)`	M	input value from PLC

Video Control Functions

Name	Type	Description
`int capture_request(int exp, int gain, int *start, int mode)`	S	Put request for image capture into capture queue
`int cancel_capture_rq(void)`	S	abort capture request queue
`void vmode(int mode)`	C	Set video modes
`void tpict()`	C	Picture taking function
`long shutter(long stime)`	C	Select shutter speed
`int tpp(void)`	C	Picture taking function for progressive scan
`int tpstart(void)`	C	Picture taking function for progressive scan
`void tpwait(void)`	M	Wait for completion of picture taking function / progressive scan
`int tenable(void)`	C	Trigger enable for interrupt driven image acquisition
`int trdy(void)`	C	Check the status of the picture taking function / external trigger mode
`void SET_trig_lossy(void)`	M	select “lossy” external trigger mode
`void SET_trig_sticky(void)`	M	select “sticky” external trigger mode

RS232 (V24) Basic Functions

Name	Type	Description
`void rs232snd(char c)`	S	Output a character/serial interface
`void putchar(char c)`	M	Output a character/serial interface
`char rs232rcv()`	S	Read a character/serial interface
`char getchar()`	M	Read a character/serial interface
`int sbready()`	S	send buffer ready/serial interface
`int rbready()`	S	receive buffer ready/serial interface
`void setbaud(long baudrate)`	S	set baudrate for serial interface
`char kbdrcv()`	S	Read a character/keyboard
`int kbready()`	S	receive buffer ready/keyboard

Utilities

Name	Type	Description
`int getvar(int var)`	S	Read system variable
`void setvar(int var, int x)`	S	Write system variable
`long getlvar(int var)`	S	Read system variable (long)
`void setlvar(int var, long x)`	S	Write system variable (long)
`float getfvar(int var)`	S	Read system variable (float)
`void setfvar(int var, float x)`	S	Write system variable (float)
`int getstptr()`	A	Read stack pointer
`int getdp()`	A	Read data pointer
`int getbss()`	A	read start of bss

Lookuptable Functions

Name	Type	Description
`int set_overlay_bit(int bit, int r, int g, int b)`	C	assign a color to an overlay bitplane
`void set_translucent(int table, int r, int g, int b)`	C	assign a color to a translucent overlay table
`void set_ovlmask(int mask)`	C	set overlay mask register
`void init_LUT(void)`	C	init image data LUT / black-and-white
`void init_LUT_gamma(float gamma)`	C	init image output LUT using gamma correction
`void init_color_lut(I32 red, I32 green, I32 blue)`	C	initialize color input LUT

Time related functions

Name	Type	Description
`void c_time(long zsec, int tz, int *sec, int *min, int *hour)`	C	convert system time – extract time
`void c_date(long zsec, int tz, int *day, int *month, int *year)`	C	convert system time – extract date
`void c_timedate(long zsec, int tz, int *sec, int *min, int *hour, int *day, int *month, int *year)`	C	convert system time – extract date and time
`void ltime(int *sec, int *min, int *hour)`	M	convert system time – extract local time
`void ldate(int *day, int *month, int *year)`	M	convert system time – extract local date
`void ltimedate(int *sec, int *min, int *hour, int *day, int *month, int *year)`	M	convert system time – extract local date and time
`void gtime(int *sec, int *min, int *hour)`	M	convert system time – extract GMT time
`void gdate(int *day, int *month, int *year)`	M	convert system time – extract GMT date
`void gtimedate(int *sec, int *min, int *hour, int *day, int *month, int *year)`	M	convert system time – extract GMT date and time
`unsigned long x_timedate(int tz, int sec, int min, int hour, int day, int month, int year)`	C	calculate system time
`void xtimedate(int sec, int min, int hour, int day, int month, int year)`	M	calculate system time and system store in variable SEC

Legend: A: Assembly function C: C function S: System call M: Macro

Index

Visit the Vision Components site www.vision-components.com for further information and documentation and software downloads:

Web Site Menu Links	Content
Home	Latest News from VC
Our Company	VC Company Information
Contact Us	Distributor list / Enquiry forms
News	More News form VC
Products VC Smart Camera Hardware VCXX Camera Series VC20XX, VC4XXX Smart Cameras VCSBC Board Cameras VCM Camera Sensors VC Smart Camera Software **VCRT Operating System** **VCLIB Image Processing Library** **Vision Components' Special Libraries** **VC Smart Camera Accessories**	Product Overviews: including accessories listings with corresponding order numbers M200 Data Matrix Code Reader VCOCR Text Recognition Color Lib Cables, lenses and other accessories
Support: **Support News**	Overview about latest features, manuals and SW updates
Knowledge Base / FAQ	Searchable HW and SW information database
Download Area Public Download Area **(free access)** Registered User Dl Area **(registration required)** Customer Download Area **(user- and software registration required)**	Download of all: - Product brochures - Camera Manuals - Getting Started - Software Manuals - Training Manuals and Code - Developement Libraries and Camera OS Updates and Archives - Special Library Updates - Utility Software
RMA Number Request Form	- Repair Number Request Form

Vision Components GmbH
Ottostraße 2 • D-76275 Ettlingen
Tel. +49(7243)2167-0
Fax +49(7243)2167-11
sales@vision-components.de