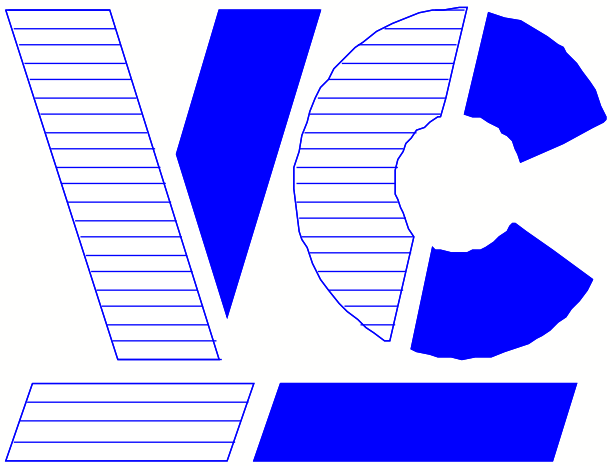


Vision Components

The smart camera people ...

Vision Components Software Documentation Version 5.0

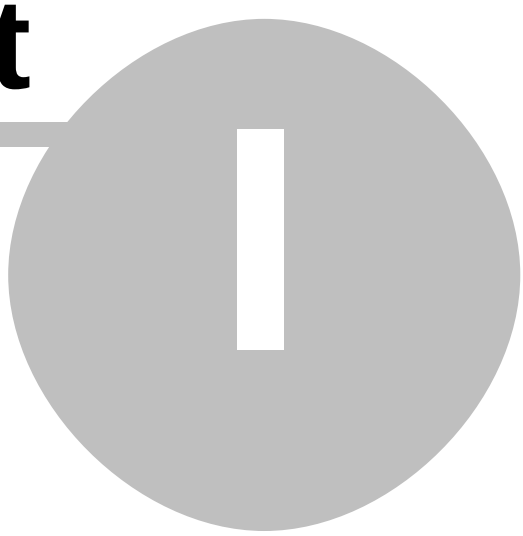
August 2003

Table of Contents

Foreword

This documentation was created very conscientiously. No liability is assumed for possible errors or misleading descriptions. The information contained in this documentation is informative and in no way guarantees the characteristics of the product. The right is reserved to make technical changes dictated by the state of the art.

Part

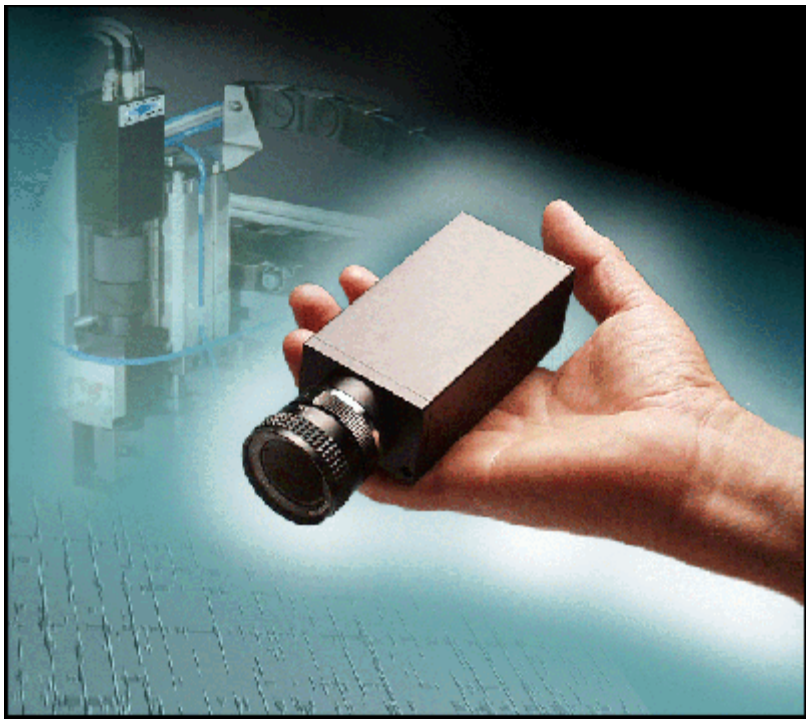


1 Introduction

Preleminary !!! 0.71

Software Documentation
VC Series Machine Vision Cameras
Operating System VC/RT
General Library Functions

Version 5.0x



misleading descriptions which it may contain. The statements made in this documentation are informative in nature and not a guarantee of features. The right is reserved to make changes in the interest of technical progress.

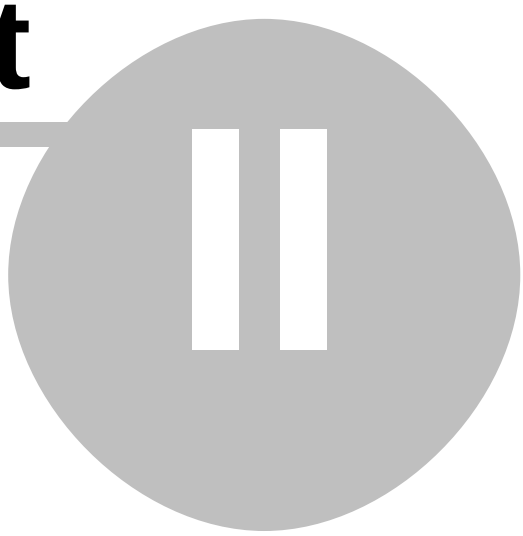
This documentation describes the VC/RT operating system software version 5.0x.

You can also consult the following documentation:

- Hardware documentation Hardware
- Documentation VCLIB Image Processing Library

Caution: VC/RT 5.0x only runs on VC-smart cameras with TMS32C62xx processor.

Part



2 General Information

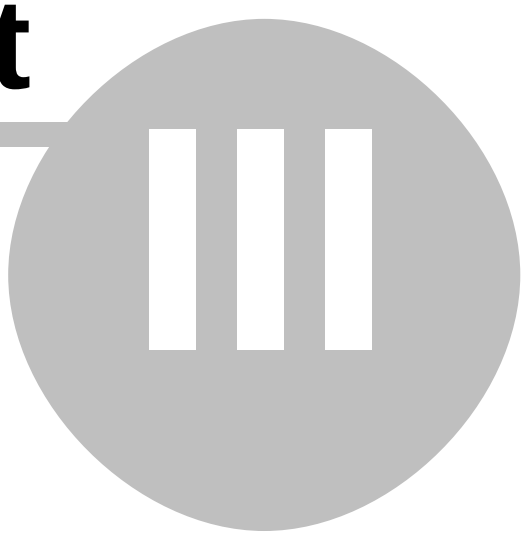
The VC Series cameras are compact, light-weight black-and-white or color video cameras with video memory and a frame processor.

They integrate a high-resolution CCD 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.

Part



3 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 also supports the user in the debugging and test phase.

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

Property	VC/RT	MS-DOS	OS/9
Real-time capable	yes	no	yes
Multitasking	yes	no	yes
License	one-time*)	per installation	per installation

*) per developer workstation

Part IV

4 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 administer somewhat "uncommon" resources:

Resource	Functions
CCD sensor	Picture taking and reproduction, various control functions
Frame output	Control of the display and overlay outputs
Flash EPROM	Loading and saving files or programs, deleting sectors
multi-media card	File access
SDRAM	Accessing and managing memory, allocating and releasing memory
RS232 interface	Data buffering and background I/O operations
Ethernet	Full Highspeed TCP/IP stack
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
- Various routines which can be linked to the user program

Part V

5 The VC/RT Kernel

The kernel is located permanently at addresses 0xA0000000 through 0xA001FFFF in SDRAM.
It thus occupies 128 kBytes of memory. (The memory model is described in Organization of the DRAM 176)

The kernel consists of the following components:

- During power-up or reset, the loader loads the shell (filename: "shell"). The continually resident routine "exec()" can be used to dynamically load programs at any time.
- Interrupt-controlled routines for time management. Via an interrupt, all time-related functions are controlled once per msec.
- 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. The update frequency of the display refresh memory is programmable to once per each video frame or any multiple of the frame rate.
- EDMA-controlled routines for multi-media card access

Part VI

6 The Shell ("shell")

The shell is a program loaded by the loader. The shell communicates with the user via the serial interface. (A PC with a communications program, such as PROCOMM, is commonly used for this. PROCOMM is discussed below.)
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 are stored as a file on the flash EPROM. (The command could also be a user program, 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 maybe 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.

6.1 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** >]
dd	DMEM Display	dd <**addr** >||<range>
dwn	download file to PC	dwn <**path** >
er	erase **complete** flash eprom	**er**
ex	**exit from shell**	**ex**
fd	**multi-media card display**	**fd <addr>||<range>**
?	help	? [<name>]
he	help	he [<name>]
help	help	help [<name>]
ht	Hardware Test	ht
js	jpeg store	js <**path** >
jl	jpeg load	jl <**path** >
jt	jpeg transfer	jt
lo	Load S Records	lo [<**option** >]
mem	**display memory usage**	mem [<**option** >]
pk	pack flash memory	pk
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>]

6.1.1 Shell Command "bd"

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 baud rate 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

6.1.2 Shell Command "cd"

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)

6.1.3 Shell Command "cx"

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.)

6.1.4 Shell Command "copy"

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.

ertewrt

examples

copy md:/my_directory/test.jpg	copies test.jpg from directory "my_di MMC to current data directory
copy fd:test.jpg md:/test.jpg	copies file test.jpg from flash to MM

6.1.5 Shell Command "del"

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".
Note: 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.

6.1.6 Shell Command "dir"

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:

6.1.7 Shell Command "dwn"

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.

6.1.8 Shell Command "er"

er **erase sector / flash EPROM**

synopsis er

description The entire flash EPROM can be physically erased (formatted) with the command er (except for sector 0). 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 107() is still available. Please use file based functions 57 instead

6.1.9 Shell Command "ex"

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.

6.1.10 Shell Command "he"

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 16 is included as a parameter, **he** displays the syntax for the corresponding command.

6.1.11 Shell Command "ht"

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"

6.1.12 Shell Command "jl"

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

6.1.13 Shell Command "js"

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

6.1.14 Shell Command "jt"

jt **jpeg transfer**

synopsis jt

description Entering jt will transfer the complete image of the frame buffer (memory page 0) to the V24 / RS232 serial port, resp. to the telnet port (port 23) of an Ethernet camera. Every 1024 characters a character "A" is expected as an acknowledge. Every other character will cause a retransmit of the 1024 bytes. NO characters will cause the system to hang.
The Graphic Shell has an image download feature included.

On VC's Ethernet cameras you may use the ftp feature to transafer a jpeg from and to the PC.

6.1.15 Shell Command "lo"

lo **load S Records / flash EPROM**

synopsis lo [<option>]

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. (see also the corresponding description of PROCOMM)
lo is usually called without option, which defaults to "hex mode".
When the loading is done, **lo** will delete all older files with the same name and type.

On Ethernet cameras ftp can be used instead

possible options -h hex mode (default) for use with PROCOMM

6.1.16 Shell Command "mem"

mem **display memory usage**

synopsis mem [<option>]

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

Entering mem without option will display the usage of all memories.

Options:
-t display 'text' memory segment usage
-s display 'stack' memory segment usage
-d display 'data' memory segment usage

-i display 'image' memory segment usage
-f display flash memory usage

version 5.08 and earlier : not implemented yet

6.1.17 Shell Command "pk"

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 DRAM available. This will happen if DRAM memory was allocated by a program, but not freed.

6.1.18 Shell Command "time"

time **display system time**

synopsis time [<option>]

description VC/RT for VC20xx 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:

-t display 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

examples :

time
time and date command
temperature: 54.0 C
current timezone: +01

daylight savings time: ON
time: 14:55:20
date: 12/31/00

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

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

6.1.19 Shell Command "tp"

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**

6.1.20 Shell Command "type"

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
```

6.1.21 Shell Command "sh"

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.

6.1.22 Shell Command "ver"

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 5.08
```

6.1.23 Shell Command "vd"

vd **set video modes**

synopsis

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

description

The video modes can be changed with vd. There are the following options:

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.

note:

different from the VCxx cameras on the VC20xx cameras live mode always stores the image in memory.
This is valid esp. for `vmode(0)`.

Part

7 Supplied Utilities

A series of PC utilities are included. They are described below.

7.1 Procomm

PROCOMM is a data communications program with terminal emulation. It is used to communicate with the camera via the serial interface and to transfer data and programs from a PC to the video camera (or vice versa). PROCOMM is a shareware program (see the built-in copyright notice, which can be called by pressing ALT I). It should be mentioned that a "professional" version is available from retail stores. There are also a number of other products with similar functions which may be usable.

7.1.1 Important Key Combinations for Procomm

PROCOMM has numerous options. Only the most important ones will be described here.

The following key combinations are important when working with PROCOMM:

Enter	Function	Description
ALT-F10	Help	all possible key combinations are displayed
ALT-P	Modem Parameters	the baud rate and other transmission parameters can be changed here
ALT-S	Procomm Setup	important settings can be changed here, especially regarding the nature of the emulated terminals and the transfer of ASCII data
ALT-X	Exit	PROCOMM is exited with this command

7.1.2 Settings for Procomm

The basic settings for PROCOMM are the selection of the baud rate and the port for the PC serial interface.

Determine which PC port you will be using to communicate with the camera (COM1:, COM2:, etc.). Connect the camera's V24 cable to the 9-pin or 25-pin plug of your PC's serial interface (COM1:, COM2:, etc., depending on the port you choose).

You may have to solder an appropriate 9-pin or 25-pin plug, in accordance with the pin assignments specified in the hardware description. Or you may have to use a 9-to-25-pin plug adapter.

Start PROCOMM (call: PROCOMM <ret>). Enter ALT P. You will see the menu "COMMUNICATION PARAMETERS" for choosing a port and the baud rate. Use the listed numbers to select the menu positions, until you see the desired setting in the top line, "CURRENT SETTINGS". At the factory, the cameras are set to 9600 baud. So a correct setting might be as follows:

```
9600,N,8,1,COM2
```

The setting for the number of bits, the parity and the number of stop bits is fixed and should not be changed

```
Number of bits:          8
Number of stop bits:     1
Parity:                  none
```

Only the PC port and the transmission rate can be changed. If you change the transmission rate, you must first change this setting for the camera (command bd).

After you have made the correct settings for PROCOMM and have saved them, you can power up the camera.

The copyright messages for the loader and the shell must appear on the screen now.
If they do not, you may have configured the wrong port, the cable for the V24 interface may be defective or incorrectly soldered, or the camera is possibly not powered up.
If random characters are displayed, the baud rate, parity or stop bits are probably set incorrectly. Random characters can also result from incorrect soldering of the V24 plug.

If the messages do appear on the PC screen, please hit <return> a few times to check the link between the PC and the camera. The prompt ($) must appear each time.

7.1.3 Uploading and Downloading with Procomm

In addition to sending and receiving characters, PROCOMM can send entire files from the PC to the camera (uploading) or from the camera to the PC (downloading). Uploading is especially important, in order to transfer programs created at the PC to the camera.

The key "Page Up" activates the upload/download function.
The key "Page Down" activates the download function.
PROCOMM then queries you for the transmission protocol. Select ASCII protocol (menu item 7) for both cases. Finally, PROCOMM queries you for the filename.

For uploads, this is usually the .MSF file to be sent to the camera. For downloads, a file is created at the PC with this name. The received data will be stored in this file.

Note: Uploading files using the lo-command requires handshaking. Using XON/XOFF handshaking is recommended. (PROCOMM: ALT-S | 2) TERMINAL SETUP | 3) Handshake ... XON/XOFF)

7.2 ECONV

The program ECONV converts the output of the linker (COFF-compatible .out file) to a VCRT file.

The C source code is usually compiled and then linked with the linker. ECONV then reads the .OUT file which has been created by the linker. Unlike the similar CONVERT utility used for Analog Devices DSPs, ECONV only produces 1 module.

The module thus created is "wrapped" in a file structure. The resulting file needs only be transferred to the camera. This file contains all relevant information for the file system and the camera loader.

ECONV is called with a parameter.

This parameter is the filename to appear in the directory of the camera. It may be at most 8 characters long.

Example:

```
ECONV pgm1
```

ECONV uses fixed file names for the input and output files, namely "EXEC.OUT" for the input file and "ADSP.OUT" for the output file.

7.3 ACONV

The function of ACONV is similar to that of CONVERT. However, ACONV works with ASCII input files. Like CONVERT, ACONV has a parameter which specifies the file name for the camera's directory.

With ACONV, the fixed name for the input file is "ASCII.INP". The output file is named "ADSP.OUT".

Example :

```
ACONV text1
```

7.4 BCONV

The function of BCONV is similar to that of ACONV. However, BCONV works with BINARY input files. Like ACONV, BCONV has a parameter which specifies the file name for the camera's directory.

With BCONV, the fixed name for the input file is "BINARY.DAT". The output file is named "ADSP.OUT".

Example :

```
BCONV dat1
```

7.5 JCONV

The function of JCONV is similar to that of BCONV. JCONV uses a gray level JPEG image as input. However, JCONV will produce an output file with filetype=3 (JPEG). Please note, that grey-value JPEG files are supported only.

With JCONV, the fixed name for the input file is "BINARY.DAT". The output file is named "ADSP.OUT".

Example : `JCONV img1`

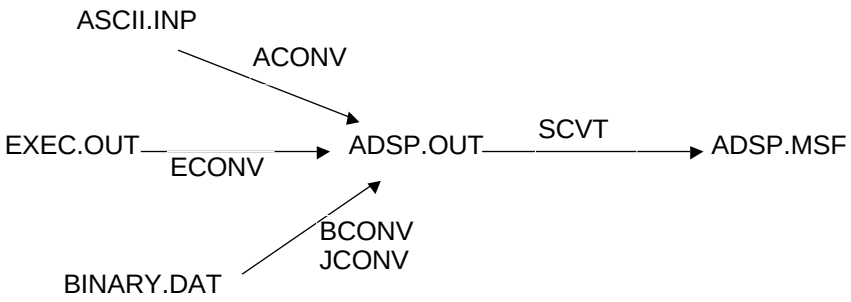
7.6 SCVT

Before the files created with CONVERT or ACONV can be sent to the camera, they must be converted to a so-called S-Record. The program SCVT is used for this. In addition to the useful information, S Records contain check sums and load addresses. Thus, transmission errors can be recognized immediately.

SCVT is called without parameters.

The input file is always named "ADSP.OUT", while the output file is named "ADSP.MSF".

7.7 Diagram of the Utilities



7.8 SMERGE

The program SMERGE makes it possible to merge two S-record format files (.MSF files) as one file. Executing this program repeatedly allows the user to create files from virtually any number of individual files.

A file created with SMERGE can be sent as usual via the serial interface to the camera.
As always, the VC/RT files are in the camera after the upload. The command dir, for example, can be entered to check for their presence.

The reason for this program is that at the PC, all camera programs can be merged as a single file. This greatly simplifies the process of installing the camera software.

Call syntax:

```
SMERGE file1 file2 outputfile
```

file1 and file2 are the two S-record files which are merged, outputfile is the merged file.

outputfile must not be identical with either of the files file1 or file2.

Example:

```
SMERGE pgm1.msf pgm2.msf allprog.msf
```

7.9 S2B

This utility converts S-records to binary files. One application might be the transfer of JPEG-files from the camera to the PC using S-records. S2B may then be used to convert these data to a binary JPEG-file which may be viewed with standard PC-programs.

S2B is called without parameters. The input file is always named "INPUT.MSF", while the output file is named "OUT.DAT".

7.10 VCINIT.BAT

Before starting a development for the VC series cameras, the development system must be configured for the particular camera model. The main differences of the various camera models are as follows:

- resolution and type of the CCD sensor (interlace / progressive scan)
- speed / clock frequency of the processor

•size (pagesize, number of pages), speed and access type (conventional / pipelined) of the DRAM
•size (number of sectors), speed of the flash Eprom

The configuration is done using VCINIT.BAT

at the DOS command line you enter the following:

```
vcinit xxx
```

where xxx is the camera model you use

example:

```
C:\ADSP\21XX\WORK>vcinit xxx
```

VCINIT.BAT copies a number of files, which are neccessary for the development system:

NOTE : VCINIT is not available for VC/RT 4.0 yet.

Important Notice:

If you start developing for a different camera model, please be sure to use VCINIT before, otherwise the compiled and linked program running on the camera may "hang" when accessing memory, picture acquisition, etc.

Part

8 The File System

The cameras have a **FLASH EPROM** which is used in a way similar to how hard disks are used in larger computers.
Programs and data can be stored permanently here and can be reloaded at any time.

In contrast to many operating systems for larger computers, there is **only one directory without subdirectories** .

Files are given a name, which can be up to 8 ASCII characters. There is no extension, but an equivalent designation can be made part of the name. ("prog.exe", for instance, is a valid file name, while "prog1.exe" is not, because it includes more than 8 characters.)

The flash EPROM has 32 sectors of 64 KB each. Sector 0-5 are the boot sectors with the operating system and some standard utilities. Sectors 0-5 are read-only. They cannot be deleted, nor can the data be overwritten. (The reason for this is that the camera could malfunction if parts of the operating system were to be overwritten. The camera would have to be sent in for "repairs".)

The remaining 26 sectors (sectors 6 through 31) are for the user. All functions (reading, deleting and writing) are possible here.

A special aspect of the flash EPROM is that a sector must first be deleted before data can be written to it. The sector structure is irrelevant when data is written or read. This means that files can be larger than one sector.

The VC20xx series of cameras except VC2028 also have a 16 Megabyte **Multimediacard (MMC)** that can be used for program and data storage. Different from the Flash Eprom, the MMC allows a directory tree, say subdirectories.

Paths of files on the flash EPROM start with **"fd:/"**
Paths of files on the MMC start with **"md:/"**

Files can be copied from one to the other storage medium by the **copy** command:

example :

```
copy fd:/test.exe md:/test.exe
```

8.1 Loading Programs to the Flash EPROM

Programs are loaded with the shell command **lo** (load S-Record file).

After you enter the command (**lo**), the program waits for valid S Records. So at

this point, please do not make any inputs by hand but rather send an S record file.

This is best done with the upload function of a communications program, such as PROCOMM, TELIX, etc.
The S-Record file itself is created from the .EXE file of the Analog Devices operating system, using the programs CONVERT and SCVT. The required file structure is created during this process.

CONVERT is called, with the name of the file to be created as the parameter.

Example :

Input file: EXEC.OUT

Enter: ECONV myprog
Enter: SCVT

Upload: ADSP.MSF

At the camera, the program can then be called with the name "myprog".

It is NOT possible to load the program to other locations than the flash EPROM. Say, a path "fd:/myprog" is implied.

If you want to copy a program to the MMC, first load it to the flash EPROM , then copy it to the MMC with the copy command.

It is NOT possible to directly load a program into DRAM without storing it to the Flash EPROM.

Part

9 The Operating System Function "exec"

The operating system call **exec** () can be used to dynamically postload programs from the flash EPROM or MMC 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 subroutines
Postloading requires CPU time	All subroutines are always available immediately

Postloading is very recommendable if the program was structured for this and each partial program contains different functions. For instance, a command interpreter and the called commands could be organized this way.
This technique is possible, but not always recommendable, when the partial programs mostly contain the same subroutines with few differences.

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

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

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

Note: 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

Programs are usually loaded starting at memory address 0xA0200000. All user programs including the shell and all programs called by exec are loaded this way.

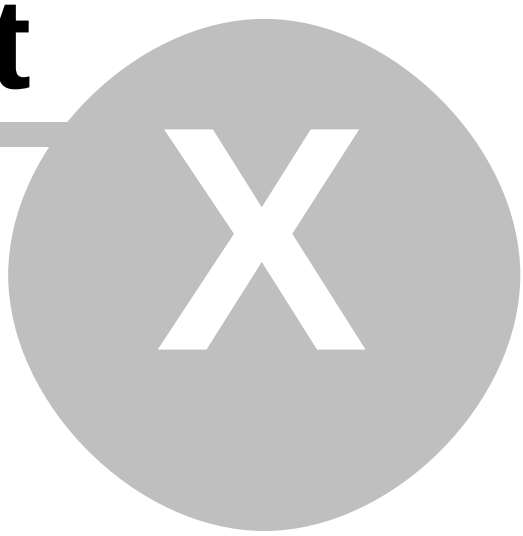
Advanced users may change the *.cmd file to load programs to a different address.

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)**

Part

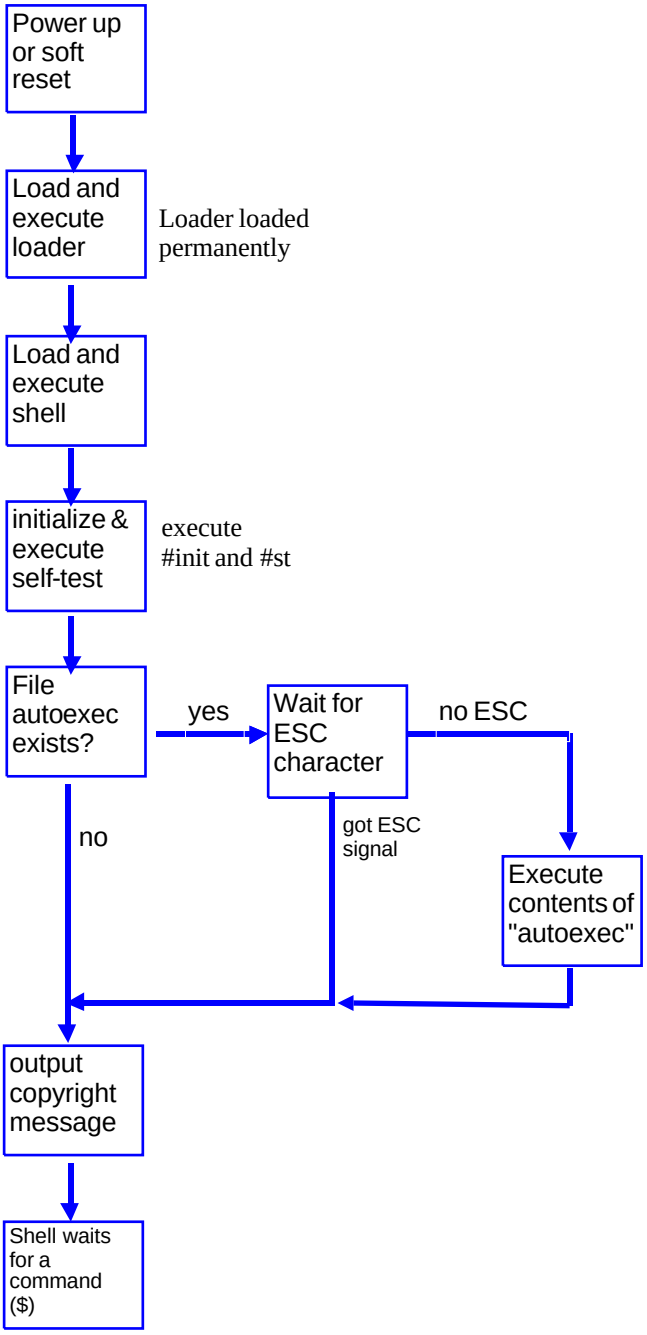


10 Auto Execution of Programs when booting

The cameras are used in industrial automation. Thus, at least the user program must be executed automatically when booting (powering up).

The ASCII file "autoexec" can be used, like for PC systems. The commands and programs it contains are interpreted by the shell one after another and executed line by line. The file "autoexec" can be created on the development system (PC). For this, the file "ASCII.INP" is edited. The conversion tools "ACONV" (enter the command: "ACONV autoexec") and "SCVT" are used. Finally, the .MSF file thus created is transferred to the camera's flash EPROM with the command **lo**.

The system boots as follows:



Part

11 Descriptions of the Library Functions

If needed, the library functions described below can be linked to any C program.

Some of the functions have different versions for each camera model, others are available only for a specific camera model.

Please make sure to use the appropriate configuration with the VCINIT batch utility.

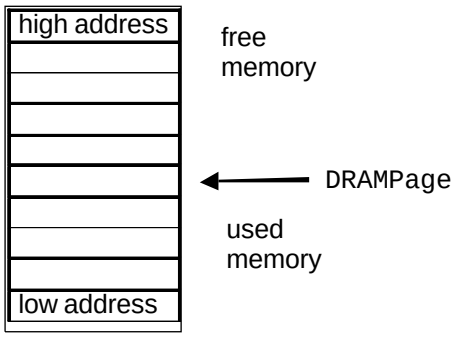
11.1 Overview of the Library Functions

11.2 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.

vcsetup 48	Initialize memory management
vcmalloc 49	user memory allocation
vcfree 49	user memory release
prtfree 50	print a list of available user memory segments
sysmalloc 50	system memory allocation
sysfree 51	system memory release
sysprtfree 52	print a list of available system memory segments

For the allocation of DRAM memory space a very simple allocation scheme is used. Clustersize for the allocation is one DRAM page (1024 or 2048 words depending on the memory used). A pointer is used which points to the first available DRAM page. Pages below this pointer are in use, pages above and equal to DRAMPage are free. Allocating and releasing parts of the memory means moving up and down the pointer. On power-up the system allocates memory for one video and one overlay frame.



DRAMPagesAvail 52	number of available DRAM pages
DRAMBytesAvail 52	number of available DRAM bytes
DRAMWordsAvail 53	number of available DRAM words
DRAMPgMalloc 53	allocate DRAM memory in units of a memory page
DRAMPageMalloc 53	allocate DRAM memory in bytes, return start page of block
DRAMByteMalloc 54	allocate DRAM memory in bytes, return start byte-address
DRAMWordMalloc 54	allocate DRAM memory in words, return start word-address
DRAMByteFree 54	return memory block to DRAM allocation system (byte-address)
DRAMWordFree 55	return memory block to DRAM allocation system (word-address)
DRAMPgFree 55	return memory block to DRAM allocation system (page-address)
DRAMScreenMalloc 56	allocate DRAM memory for full screen storage
DRAMOvlMalloc 56	allocate DRAM memory for full screen overlay storage

11.2.1 vcsetup

vcsetup **Initialize memory management (macro)**

synopsis void vcsetup(void)

description **vcsetup** () was used in previous versions of VC/RT to initialize memory management. This is however **not necessary any more**, since the operating system takes care of the memory initialisation on power-up.

This function has been maintained for reasons of compatibility as a macro.
However: **calling vcsetup() has no effect.**

11.2.2 vcmalloc

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.

The heap is located in the **data memory segment** , so the allocated memory areas can be used as buffers for block transfers, e.g. for the routines blrdw 76(), blrdb 81(), blwrw 77(), etc.

vcmalloc ()is basically equivalent to the function **malloc** (), which most systems provide as a runtime library function.

However, the use of malloc() from the runtime library of the cross-development system by Texas Instruments is not recommended.

see also vcfree 49(), sysmalloc 50()

11.2.3 vcfree

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.

However, the function free() from the runtime library of the cross-development system by Texas Instruments should not be used.

example

```
#include <vclib.h>

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

see also vcmalloc 49(), sysmalloc 50()

11.2.4 prtfree

prtfree **print a list of available user memory segments (macro)**

synopsis void prtfree(void)

description The function **prtfree** () outputs a list of the available memory segments of the heap via the serial interface, resp. Telnet (port 23 of the Ethernet).

This can be a useful programming tool, especially in the test phase.

see also vcmalloc 49(), vcfree 49(), sysprtfree 52()

11.2.5 sysmalloc

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.

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 49 (), sysfree 51 ()

11.2.6 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 49 (), sysmalloc 50 ()

11.2.7 sysprtfree

sysprtfree **print a list of available system memory segments**

synopsis void sysprtfree(void)

description The function **sysprtfree** () outputs a list of the available memory segments of all SDRAM memory segments.

This can be a useful programming tool, especially in the test phase.

see also sysmalloc 50 (), sysfree 51 ()

11.2.8 DRAMPagesAvail

DRAMPagesAvail **number of available DRAM pages**

synopsis int DRAMPagesAvail(void)

description **DRAMPagesAvail** () returns the number of available DRAM pages of the DRAM allocation system.
Note that the total number of DRAM pages as well as the DRAM pagesize may differ for the various camera models.

11.2.9 DRAMBytesAvail

DRAMBytesAvail **number of available DRAM bytes**

synopsis long DRAMBytesAvail(void)

description **DRAMBytesAvail** () returns the number of available DRAM bytes of the DRAM allocation system.
Note that the value returned is a multiple of the number of bytes per page, since the memory is allocated in units of one DRAM page.

11.2.10 DRAMWordsAvail

DRAMWordsAvail number of available DRAM words

synopsis long DRAMWordsAvail(void)

description **DRAMWordsAvail** ()returns the number of available DRAM words (16 bits) of the DRAM allocation system.
Note that the value returned is a multiple of the number of words per page, Since the memory is allocated in units of one DRAM page.

11.2.11 DRAMPgMalloc

DRAMPgMalloc allocate DRAM memory in units of a memory page

synopsis int DRAMPgMalloc(unsigned int count)

description **DRAMPgMalloc** () allocates count pages of DRAM memory and returns the start page of the allocated memory block.

If the memory size requested is not available, the function will return -1.

11.2.12 DRAMPageMalloc

DRAMPageMalloc allocate DRAM memory in bytes, return start page of block

synopsis int DRAMPageMalloc(unsigned long nbytes)

description **DRAMPageMalloc** ()allocates nbytes bytes of DRAM memory and returns the start page of the allocated memory block. Allocation is done in units of the DRAM pagesize. If nbytes is not a multiple of the pagesize, the number of pages allocated is rounded up.

If the memory size requested is not available, the function will return -1.

11.2.13 DRAMByteMalloc

DRAMByteMalloc allocate DRAM memory in bytes, return start byte-address

synopsis long DRAMByteMalloc(unsigned long nbytes)

description **DRAMByteMalloc** ()allocates nbytes bytes of DRAM memory and returns the start (byte) address of the allocated memory block. Allocation is done in units of the DRAM pagesize (number of bytes per DRAM page). If nbytes is not a multiple of the pagesize, the number of pages allocated is rounded up.

If the memory size requested is not available, the function will return -1L.

11.2.14 DRAMWordMalloc

DRAMWordMalloc allocate DRAM memory in words, return start byte-address

synopsis long DRAMByteMalloc(unsigned long nwords)

description **DRAMByteMalloc** ()allocates nwords words (32 nit each) of DRAM memory and returns the start (word) address of the allocated memory block. Allocation is done in units of the DRAM pagesize (number of bytes per DRAM page). If nwords is not a multiple of the pagesize, the number of pages allocated is rounded up.

If the memory size requested is not available, the function will return -1L.

11.2.15 DRAMByteFree

DRAMByteFree return memory block to DRAM allocation system (byte-address)

synopsis void DRAMByteFree(long startbyte)

description The function **DRAMByteFree** ()is used for returning unused DRAM memory blocks to the DRAM memory allocation system. The

startbyte address of the block is simply passed to the function.
Note, that there is always a coherent block of memory being used directly underneath the (coherent) free memory area.
This means, that it is only possible to return the last recently allocated memory block. If DRAMByteFree() is called with an address of a memory block allocated earlier, it will free all memory blocks which have been allocated in the meantime down to this very address.

11.2.16 DRAMWordFree

DRAMWordFree **return memory block to DRAM allocation system (word-address)**

synopsis void DRAMWordFree(long startword)

description The function **DRAMWordFree** ()is used for returning unused DRAM memory blocks to the DRAM memory allocation system. The startword address of the block is simply passed to the function.
Note, that there is always a coherent block of memory being used directly underneath the (coherent) free memory area.
This means, that it is only possible to return the last recently allocated memory block. If DRAMWordFree() is called with an address of a memory block allocated earlier, it will free all memory blocks which have been allocated in the meantime down to this very address.

11.2.17 DRAMPgFree

DRAMPgFree **return memory block to DRAM allocation system (page-address)**

synopsis void DRAMPgFree(int startpage)

description The function **DRAMPgFree** () is used for returning unused DRAM memory blocks to the DRAM memory allocation system. The startpage of the block is passed to the function.

Note, that there is always a coherent block of memory being used directly underneath the (coherent) free memory area.
This means, that it is only possible to return the last recently allocated memory block. If DRAMPgFree() is called with the startpage of a memory block allocated earlier, it will free all memory blocks which have been allocated in the meantime down to this very address.

11.2.18 DRAMScreenMalloc

DRAMScreenMalloc **allocate DRAM memory for full screen storage**

synopsis int DRAMScreenMalloc(void)

description The function **DRAMScreenMalloc** ()allocates DRAM memory for one screen of video display. It returns the start page of the allocated memory block. This start page may conveniently be used to instruct the video controller to display the memory area on the video monitor.

example

```
int newpage;
newpage=DRAMScreenMalloc(); /* allocate
memory           */
setvar(stpage,newpage);  /* display new
memory area*/
```

11.2.19 DRAMOvlMalloc

DRAMOvlMalloc **allocate DRAM memory for full screen overlay storage**

synopsis int DRAMOvlMalloc(void)

description The function **DRAMOvlMalloc** ()allocates DRAM memory for one screen of video overlay display. It returns the start page of the allocated memory block. This start page may conveniently be used to instruct the video controller to display the corresponding overlay memory area on the video monitor.

example

```
int newpage;
```

```
newpage=DRAMOvlMalloc();      /* allocate
memory                          */
setvar(ovpage,newpage); /* display new
overlay area      */
```

11.3 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 58	open a device, get file pointer
io_fclose 58	close device
io_read 59	read from device
io_write 59	write to device
io_ioctl 59	control function
io_fgetc 60	get character from device
io_fputc 60	put character to device
io_fseek 61	set file position
io_get_handle 61	get a pointer to the default standard I/O stream

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 restrictions apply:

Drive	Access Mode	Operation
fd:	Read	Unlimited number of read accesses to same file
	Write	Access to only 1 file in total
md:	Read	Unlimited number of read accesses to same file
	Write	Access to file is locked for other tasks

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.

11.3.1 io_fopen

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

11.3.2 io_fclose

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 58. The function returns 0 for successful operation or otherwise an error number, which depends on the driver for the selected device.

11.3.3 io_read

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 58.
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.

11.3.4 io_write

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 58.
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.

11.3.5 io_ioctl

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
STDIN	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
fd:	IO_PACK	pack	&result
	IO_ERASE	erase	&result
	IO_READDIR	read directory	READDIR
	IO_CHKSYS	check system	NULL
	IO_DEL	delete file	NULL
md:	IO_PACK	pack directory	NULL
	IO_READDIR	read directory	READDIR
	IO_DEL	delete file	NULL
	IO_MKDIR	make directory	NULL

11.3.6 io_fgetc

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

11.3.7 io_fputc

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.

11.3.8 io_fseek

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

11.3.9 io_get_handle

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

11.4 Flash EPROM Functions

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

Some of the functions below are still available for compatibility reasons but may not be available in future versions.

The camera's flash eprom is used quite similar to a harddisk on a PC. Data is stored, too, in files. For the exact file structure, refer to **appendix D**. User files always start with flash eprom sector 1. A new file is always written right behind the last file in the file system. The file structure contains the file length, therefore a search will start at the beginning of sector 1 and parse through all files (jumping at the start of each file only) until the either the file or the end of the file system is found. Low level functions for accessing the flash eprom are discussed in chapter 10.9

11.4.1 search

search **search for a file/flash EPROM**

synopsis

```
long search(int ft, char *fname)
```

description The function `search()` looks in the flash EPROM for the file named "fname" of type "ft".

The return value is the start address of the file found.
If the file is not found, the function will return 0L.
"ft" can have the following values:

0 = executable, i.e. a program in standard COFF-format
1 = ASCII, i.e., a pure text file

2 = DATA, i.e. binary data file
3 = JPEG image file
-1: `search()` will search for the file with the specified name of
any file type

The function `search()` looks through the entire EPROM not byte for byte
but rather uses the file structure, which is much faster.

see also `fnaddr()`

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.4.2 snext

snext **search for the next free area/flash EPROM**

synopsis long snext(void)

description The function snext() looks for the next free area in the flash EPROM.
The return value is the address of this free area.
Files can be stored from this start address to the end of the flash EPROM.

see also fnaddr 63()

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.4.3 fnaddr

fnaddr **search for the start address of the next file/flash EPROM**

synopsis long fnaddr(long addr)

description The function fnaddr() calculates the start address of the next file in the flash EPROM.
The start address of a file is entered as addr.

The function then returns the address of the next file or free area.
This function returns 0L if a file header could not be found for the specified address. In particular, this is the case when addr points to a free area.

see also fname 64()

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.4.4 fname

fname **get name and type of a file/flash EPROM**

synopsis int fname(long addr, char *name)

description The function fname() gets the name and type of the file in the flash EPROM stored at address addr. The start address of a file is entered as addr. The function then returns the file type; the file name is stored in the string name by the function.

see also fnaddr 63()

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.4.5 del

del **delete a file/flash EPROM**

synopsis int del(int ft, char *fname)

description The function del() deletes the file specified by ft (file type) and fname (file name). If ft=-1 the function will delete any file matching fname only.

see also erase 107()

Since version 5.0x the functions below are replaced by general I/O functions [57] that are file based. We strongly recommend to use those.

11.4.6 fremain

fremain **remaining flash EPROM space**

synopsis long fremain(void)

description The function fremain() returns the remaining flash EPROM space in bytes.

Since version 5.0x the functions below are replaced by general I/O functions [57] that are file based. We strongly recommend to use those.

11.4.7 fcreat

fcreat **create a flash EPROM file**

synopsis void fcreat(long fp, char *name, int type)

description The function fcreat() creates a flash EPROM file by writing the file-header to address fp which should have been allocated before with the snext() function. The file may then subsequently be written to.

Finally it must be closed using flclose [66]().

Do not execute any other file operations like search [62]() before the flash eprom file is closed

Since version 5.0x the functions below are replaced by general I/O functions [57] that are file based. We strongly recommend to use those.

11.4.8 flclose

flclose close a flash EPROM file

synopsis void flclose(long fp, long length)

description The function flclose() closes a flash eprom file previously created by `fcreat()`.
The function must be supplied with the number of bytes written to the file.
From this, it calculates the end of the file and writes the file trailer.

Note: Do not leave any files open

Do not execute any other file operations like search() before the flash eprom file is closed

Since version 5.0x the functions below are replaced by general I/O functions 57 **that are file based. We strongly recommend to use those.**

11.4.9 exec

exec **Load and execute a program from the flash EPROM**

synopsis exec (char *fname, p1,p2, ... , pn)

description With the function exec(), programs (subroutines) are loaded from the flash EPROM to the SDRAM memory of the DSP and executed.
First, the name (char * fname) 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 search).
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 string, 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")`

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.4.10 loadf

loadf	**Load program from flash EPROM (for experienced user only !)**
synopsis	int loadf(long addr)
description	With the function **loadf** (), programs (subroutines) may be loaded from the flash EPROM into the DSP's main SDRAM memory. The function must be called with a valid start address (addr) of a program file (type=0) in flash Eprom. **loadf** () loads the data into memory to the load addresses specified inside the file. It then returns the PMEM address of the loaded program's entry point.
NOTE:	**Since most programs are linked to the same addresses in DMEM and PMEM, loading a program with loadf() will overwrite your program, which will result in a system crash.**
see also	exec 66()

Since version 5.0x the functions below are replaced by general I/O functions 57 that are file based. We strongly recommend to use those.

11.5 I/O Functions

pstr 69	Output a string via the serial interface
print 69	Formatted output of text and variables
sprint 70	Formatted output of text and variables to a string
hextoi 71	convert hexadecimal value string to integer
setRTS 71	set RTS signal (macro)
resRTS 71	reset RTS signal (macro)
setPLCn 72	set PLC signal (macro)
resPLCn 72	reset PLC signal (macro)
outPLC 72	output value to PLC
inPLC 73	input value from PLC (macro)

11.5.1 pstr

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 69 ()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.

11.5.2 print

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 70(), pstr 69()

11.5.3 sprint

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 69()

11.5.4 hextoi

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.

11.5.5 setRTS

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

11.5.6 resRTS

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**

11.5.7 setPLCn

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          */
```

11.5.8 resPLCn

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          */
```

11.5.9 outPLC

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.

11.5.10 inPLC

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.

11.6 DRAM Access Functions

The TMS320C62xx architecture has a huge addressing capability. SDRAM can be addressed directly using 32bit pointers. There is no need for access functions. The following "DRAM access functions" are included, however, for reasons of compatibility to VC/RT versions supporting the ADSP memory architecture.

In the ADSP architecture, DRAM memory is accessed by so-called access functions. There are functions for addressing and modification of single words and bytes (pixels) as well as block-oriented access functions which are able to copy a complete array of data to or from the DRAM at high speed. Pixel list functions are discussed in chapter 10.6

11.6.1 rd20

rd20 **Read a word from DRAM (macro)**

synopsis int rd20(long addr)

description This function allows a 16-bit word to be read from the DRAM.
The required DRAM address is handed over as a long value.

11.6.2 wr20

wr20 **Write a word to DRAM (macro)**

synopsis void wr20(int value, long addr)

description This function allows a 16-bit word to be written

to the DRAM.
The value to be written is handed over as int; the DRAM address is passed as a long value.

11.6.3 rd32

rd32 **Read a 32-bit long from DRAM (macro)**

synopsis long rd32(long addr)

description This function executes a long (32-bit) read access from DRAM.
The required DRAM address is passed as a long value.
The address 0L is the first longword in the DRAM, 2L is the second, etc.;
the last possible address depends on the size of the DRAM.

see also rd20 74()

11.6.4 wr32

wr32 **Write a 32-bit long to DRAM (macro)**

synopsis void wr32(long value, long addr)

description This function executes a long (32-bit) write access to DRAM.
The value to be written, value, and the DRAM address are passed as long values.

see also wr20 74()

11.6.5 rpix

rpix **Read a byte from DRAM (macro)**

synopsis int rpix(long addr)

description This function allows a byte to be read from the DRAM.
The required DRAM address is handed over as a long value.

The values of the addresses are thus twice as large as for word accesses to the DRAM.
The read-in byte is the LSB (bit 0 through bit 7) in the return value; the MSB (bit 8 through bit 15) is always 0.

11.6.6 wpix

wpix **Write a byte to DRAM (macro)**

synopsis void wpix(int value, long addr)

description This function allows a byte to be written to the DRAM.
The required DRAM address is handed over as a long value.
The values of the addresses are thus twice as large as for word accesses to the DRAM.
The byte to be written must be the LSB (bit 0 through bit 7) in the parameter value. For this function, the MSB (bit 8 through bit 15) can be any value; the function sets it to zero.

11.6.7 blrdw

blrdw **Read a block from DRAM, wordwise**

synopsis void blrdw(int count, int *buf, long addr)

description This function reads a block of data from the DRAM to a buffer in the DMEM of the ADSP.
The access is made wordwise, i.e., a 16-bit word in DRAM is stored as a 16-bit word in the buffer. addr is the start address of the block in the DRAM (analog to wordwise access with the function rd20()). buf is a pointer to the internal buffer in which the block is to be stored. count is the number of words to be read.

There is no restriction to the value of the pointer buf.

This function may be used when it is necessary to transfer large amounts of data as blocks. Use the function rd20 74()for random access.

11.6.8 blwrw

blwrw **Write a block to DRAM, wordwise**

synopsis void blwrw(int count, int *buf, long addr)

description This function writes a block of data from an internal buffer in the DMEM of the ADSP to the DRAM.
The access is made wordwise, i.e., a 16-bit word in the buffer is stored as a 16-bit word in the DRAM.
addr is the start address of the block in the DRAM to be written to (analog to wordwise access with the function wr20()), buf is a pointer to the internal buffer from which the block is to be read. count is the number of words to be written.

There is no restriction to the value of the pointer buf.

This function may be used when it is necessary to transfer large amounts of data as blocks. Use the function wr20 74 ()for random access.

11.6.9 blwrb

blwrb **Write a block to DRAM, bytewise**

synopsis void blwrb(int count, int *buf, long addr)

description This function reads a block of data from a buffer and writes it to the DRAM.
Access is made bytewise. That means two 16-bit words in the buffer are stored in compressed form as one 16-bit word in the DRAM.
The LSB (bit 0 through bit 7) of the first word in the buffer (lower address)
is stored as the MSB (bit 8 through bit 15) of the DRAM word. The LSB (bit 0 through bit 7) of the next buffer word is stored as the LSB (bit 0 through bit 7) of the DRAM word.
The MSB (bit 8 through bit 15) of the words in

the DMEM must always be 0, as otherwise a malfunction will result.
addr is the start address of the block in the DRAM (analog to wordwise access with the function wr20()). buf is a pointer to the buffer from which the block is to be read. count is the number of words to be written.
The number of bytes read from the buffer is twice as large as count.

There is no restriction to the value of the pointer buf.

This function may be used when it is necessary to transfer large amounts of data as blocks. Use the function wpix()for random access.

11.6.10 rovl

rovl **Read the overlay bit from DRAM (macro)**

synopsis int rovl(long addr)

description Since overlay data are stored in bytes rather than bits in the TMS320C62xx architecture this macro is essentially the same as the rpix() macro.
This function reads one byte from the DRAM. The required DRAM address is handed over as a long value. Bits 7 through 31 of the returned value are 0.

11.6.11 wovl

wovl **Write the overlay bit to DRAM (macro)**

synopsis void wovl(int value, long addr)

description Since overlay data are stored in bytes rather than bits in the TMS320C62xx architecture this macro is essentially the same as the wpix() macro.

This function writes one byte to the DRAM. The required DRAM address is handed over as

a long value. The byte to be written must be the LSB of the parameter value. For this function, bits 7 through 31 of value are redundant.

11.6.12 blrdo

blrdo **Read a block from DRAM, bitwise (overlay, macro)**

synopsis void blrdo(int count, int *buf, long addr)

description Since overlay data are stored in bytes rather than bits in the TMS320C62xx architecture this macro is mapped to the blrdb() function.

11.6.13 blwro

blwro **Write a block to DRAM, bitwise (overlay, macro)**

synopsis void blwro(int count, int *buf, long addr)

description Since overlay data are stored in bytes rather than bits in the TMS320C62xx architecture this macro is mapped to the blwrb() function.

11.6.14 xorpix

xorpix **XOR a byte in DRAM (macro)**

synopsis void xorpix(int value, long addr)

description This function executes a bytewise (8-bit) XOR write access to the DRAM.
The required DRAM address is passed as a long value.

The byte to be written must be the LSB (bit 0 through bit 7) of the parameter value. The MSB (bit 8 through bit 15) does not matter for this function - the function sets it.
The XOR function of value and the addressed pixel is calculated, and the result is written back to the same place in the DRAM.

see also wpix 76()

11.6.15 xorovl

xorovl **XOR an overlay bit (macro)**

synopsis void xorovl(int value, long addr)

description Since overlay data are stored in bytes rather than bits in the TMS320C62xx architecture this macro is essentially the same as the xorpix() macro.

see also wovl 78()

11.6.16 blrds

blrds **read block of pixels with subsampling**

synopsis void blrds(int count, int *buf, long addr, int rh)

description This function reads a block of data from the DRAM and writes it to a buffer.
It performs pretty much like the function blrdb 81(), except for that the function blrds() performs subsampling by the subsampling ratio 2*rh.

There is no restriction to the value of the pointer buf.

This function may be used when it is necessary to transfer large amounts of data as blocks.

see also blrdb 81()

11.6.17 rdrlc

rdrlc **read one line of RLC data**

synopsis int rdrlc(int dx, int *buf, long rlc)

description The function reads a line of RLC starting at DRAM address rlc
and transfers the data into DMEM starting at

address buf.
dx is the end-of-line mark of the RLC used, which is equivalent to
the horizontal width of the corresponding binary image.
The function returns the number of words transferred to DMEM as an
integer value.

There is no restriction to the value of the pointer buf.

This function may be used when it is necessary to transfer large amounts of data as blocks.

The function performs the following basic rlc input function (equivalent C-program):

```
int rdrlc(int dx, int *p, long rlc)
{
cnt=1;
while((*p=rd20(rlc++))!=dx)
  {
  p++;
  cnt++;
  }
return(cnt);
}
```

11.7 blrdb

blrdb **Read a block from DRAM, bytewise**

synopsis void blrdb(int count, int *buf, long addr)

description This function reads a block of data from the DRAM and writes it to a buffer. The access is made bytewise, i.e., a 16-bit word in the DRAM is stored in the buffer as two words (16 bits) in the buffer.
The MSB (bit 8 through bit 15) of the DRAM word is stored in the buffer as the LSB (bit 0 through bit 7) of the first word (lower address).
The LSB (bit 0 through bit 7) of the DRAM word is stored in the LSB (bit 0 through bit 7) of the next word (higher address) in the buffer.

The MSB (bit 8 through bit 15) of the words in the buffer is always 0.
addr is the start address of the block in the DRAM (analog to wordwise access with the function rd20()). buf is a pointer to the internal buffer where the block is to be stored. count is the number of words to be read.

The number of bytes written to the buffer is twice as large as count.

There is no restriction to the value of the pointer buf.

Note:
Storing the individual bytes of a 16-bit word in the LSB of the buffer is especially appropriate for frame processing functions. The contents of the buffer (such as a line from the video memory) can be modified by e.g. a filter function. The line can then be written back to the video memory using the function blwrb().

This function may be used when it is necessary to transfer large amounts of data as blocks. Use the function rpix 75()for random access.

11.8 Functions for Processing of Pixel Lists

ad_calc	address calculation for an array with x/y-coordinates
wp_list	write video memory/access via address list
wp_set	write video memory with constant/access via address list
wp_xor	XOR video memory with constant/access via address list
wo_set	write overlay with constant/access via address list
wo_xor	XOR overlay with constant /access via address list
rp_list	read video memory/access via address list
wo_list	write overlay memory/access via address list

ro_list read video memory/access via address list

11.8.1 ad_calc

ad_calc **address calculation for an array with x/y-coordinates**

synopsis

```
void ad_calc(int count, int *xy,
             long ad_list[], long start, int pitch)
```

description

This function calculates the corresponding video memory DRAM addresses for an array with x/y pairs.
The addresses can be used for the access functions rpix and wpix, as well as for the overlay functions rovl and wovl.
It is, however, especially efficient to combine ad_calc() with functions which work with address lists, such as wp_list(), rp_list(), wo_list()and ro_list().

The addresses are calculated in accordance with the following C program:

```
for(i=0; i<count; i++)
  ad_list[i] = start + (long) x[i]      + (long) y[i] *
pitch;
```

The prototype for the two-dimensional array xy[][2] is specified as int *xy. This allows various types of access (see also the examples of the function linexy()).
The arrays xy[ ][2] and ad_list[ ] are allowed to be identical. The values for x and y are then replaced by the corresponding addresses.

example

```
int pitch=getvar(vpitch);
int i,x,y,v_list[200];
long ad_list[200];
long start = 100L*pitch + 100L;
int *xy;

xy = ad_list;      /* same array */

for(i=0;i<200;i++)
 {
 x=y=i;
 xy[i][0] = x;
```

```
      xy[i][1] = y;
 v_list[i] = 255;
 }

ad_calc(200,(int
*)xy,ad_list,start,pitch);
wp_list(200,ad_list,v_list);
```

see also: Functions for processing pixel lists 202

11.8.2 wp_list

wp_list **write video memory/access via address list**

synopsis void wp_list(int count, long ad_list[], int v_list[])

description This function writes an array of values (v_list[]) to the video memory. The corresponding video memory addresses are taken from the array ad_list[].
Both arrays should be the same size, and should contain at least count elements. count is the number of pixels which are written. It must be greater than or equal to 1.

example

```
int pitch=getvar(vpitch);
int i,x,y,v_list[200];
long ad_list[200];
long start = 100L*pitch + 100L;

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]    = start + (long)x +
(long)y * pitch;
  v_list[i]     = i;
  }

wp_list(200,ad_list,v_list);
```

Note It is more efficient to use the function ad_calc() to calculate the addresses, instead of the above for loop.

see also Functions for processing pixel lists 202

11.8.3 wp_set

Enter topic text here.

11.8.4 wp_xor

wp_xor	**XOR video memory with constant/access via address list**
synopsis	void wp_xor(int count, long ad_list[], int value)
description	This function XORs the video memory with value and writes the result back to the video memory. The corresponding video memory addresses are taken from the array ad_list[]. This array should contain at least count elements. count is the number of pixels which are written. It must be greater than or equal to 1.
see also	wp_list(), wp_set(), Functions for processing pixel lists

11.8.5 wo_set

wo_set	**write overlay with constant/access via address list (macro)**
synopsis	void wo_set(int count, long ad_list[], int value)
description	This function writes value to the overlay. The corresponding overlay addresses are taken from the array ad_list[]. This array should contain at least count elements. count is the number of pixels which are written. It must be greater than or equal to 1
see also	wp_set(), wp_list()Functions for processing pixel lists

11.8.6 wo_xor

wo_xor — **XOR overlay with constant /access via address list (macro)**

synopsis — void wo_xor(int count, long ad_list[], int value)

description — This function XORs the overlay with value and writes the result back to the overlay.
The corresponding overlay addresses are taken from the array ad_list[].
This array should contain at least count elements. count is the number of pixels which are written.
It must be greater than or equal to 1

see also — wo_set, wp_xor()Functions for processing pixel lists

11.8.7 rp_list

rp_list — **read video memory/access via address list**

synopsis — void rp_list(int count, long ad_list[], int v_list[])

description — This function reads a number of pixels from the video memory and writes the corresponding values to the array v_list[].
The corresponding overlay addresses are taken from the array ad_list[].
Both arrays should be the same size and should contain at least count elements.
count is the number of pixels which are written.
It must be greater than or equal to 1.

example

```
int pitch=getvar(vpitch);
int i,x,y,v_list[200];
long ad_list[200];
long start = 100L*pitch + 100L;

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]    = start + (long)x +
(long)y * pitch;
  }

rp_list(200,ad_list,v_list);
```

```
for(i=1; i<200; i++) print("value:
%d\n",v_list[i]);
```

Note: It is more efficient to use the function ad_calc() to calculate the addresses, instead of the above for loop.

see also wp_list 84(), ro_list 88()
Functions for processing pixel lists 202

11.8.8 wo_list

wo_list **write overlay memory/access via address list**

synopsis void wo_list(int count, long ad_list[], int v_list[])

description This function writes an array of values (v_list[]) to the overlay memory. The corresponding overlay addresses are taken from the array ad_list[].
Both arrays should be the same size and should contain at least count elements.
count is the number of pixels which are written. It must be greater than or equal to 1.

example

```
int pitch=getvar(vpitch);
int i,x,y,v_list[200];
long ad_list[200];
long start;

start = (long) Overlay_Page * PGSIZE *16;
start += Overlay_Offset;

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]    = start + (long)x +
(long)y * pitch;
  v_list[i]     = 1;
  }

wo_list(200,ad_list,v_list);
```

Note: It is more efficient to use the function ad_calc() to calculate the addresses, instead of the above for loop.

see also wp_list 84(), ro_list 88() Functions for

processing pixel lists

11.8.9 ro_list

ro_list **read video memory/access via address list**

synopsis void ro_list(int count, long ad_list[], int v_list[])

description This function reads a number of pixels (1 bit) from the overlay memory and writes the corresponding values to the array v_list[].
The corresponding overlay addresses are taken from the array ad_list[].
Both arrays should be the same size and should contain at least count elements.
count is the number of pixels which are written. It must be greater than or equal to 1.

example

```
int pitch=getvar(vpitch);
int i,x,y,v_list[200];
long ad_list[200];
long start;

start = (long) Overlay_Page * PGSIZE *16;
start += Overlay_Offset;

for(i=0;i<200;i++)
  {
  x=y=i;
  ad_list[i]    = start + (long)x +
(long)y * pitch;
  }

ro_list(200,ad_list,v_list);

for(i=1; i<200; i++) print("value:
%d\n",v_list[i]);
```

Note It is more efficient to use the function ad_calc() to calculate the addresses, instead of the above for loop.

see also rp_list 86 (), wo_list 87 ()
Functions for processing pixel lists 202

11.9 Video Control Functions

11.9.1 capture_request

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 (video double-lines)
gain gain setting for ADC
start start address for image storage
mode internal / external trigger mode (mode=0 : int., mode=1 : ext.)

Exposure time is calculated according to the following formula:

Exptime[msec] = (exp + 520/944) * EXUNIT

So, exp=0 means a shutter time of approximately 30 msecs. 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

Due to the hardware architecture of the ADC

steps at gain=63/64 and gain=63/64 may not be monotonic. 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 acquisition. The system may wait indefinitely in this state if no trigger is received.

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 !

11.9.2 vmode

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.

11.9.3 tpict

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.
It does not completely support, however, 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.

Note **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.

11.9.4 tpp

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.

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

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

example

```
vmode(1);          /* still mode
     */

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

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

11.9.5 tpstart

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.

Note : Using this function does not support parallel processing (exposing while storing the image). For maximum performance, the function capture_request 89() is recommended.

11.9.6 tpwait

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 95() is completed.
If so, the function immediately returns, if not, the function waits in a loop.

11.9.7 tenable

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 signal IN0.
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 93() 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.

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

Note: Using this function does not support parallel processing (exposing while storing the image). For maximum performance, the function capture_request 89() is recommended.

11.9.8 trdy

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

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

11.9.9 shutter

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.

Note 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

11.9.10 SET_trig_lossy

SET_trig_lossy **select "lossy" external trigger mode**

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()).

11.9.11 SET_trig_sticky

SET_trig_sticky **select "sticky" external trigger mode**

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()).

11.10 RS232 (V24) Basic Functions

rs232snd 98	output a character/serial interface
rs232rcv 99	read a character/serial interface
sbready 99	send buffer ready/serial interface
sbfull 100	send buffer full/serial interface
rbready 100	receive buffer ready/serial interface
rbempty 101	receive buffer empty/serial interface
setbaud 101	set baudrate for serial interface
kbdrcv 102	read a character/keyboard
kbready 102	receive buffer ready/keyboard

11.10.1 rs232snd

rs232snd **Output a character/serial interface**

synopsis void rs232snd(char c)

description This function outputs one buffered ASCII character via the serial interface.
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.

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 255

characters.

Note : neither hardware nor software handshaking is used in this routine

see also rs232rcv 99(), sbready 99()

11.10.2 rs232rcv

rs232rcv **Read a character/serial interface**

synopsis char rs232rcv(void)

description This function reads one buffered ASCII character from the serial interface.

A background interrupt routine writes the character to the buffer. Characters will be lost if the 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.

The receive buffer can hold a maximum of 255 characters.

Note : neither hardware nor software handshaking is used in this routine

see also rs232snd 98(), rbready 100()

11.10.3 sbready

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 98(), sbfull 100()

11.10.4 sbfull

sbfull **send buffer full/serial interface**

synopsis int sbfull(void)

description This function checks the send buffer for the serial interface.
If it is full, the function returns the value -1; otherwise, it returns the amount of free space, as a number of characters.

Note: **Do not use this function for new development, since it is included for compatibility only. Use the function sbready 99() instead.**

see also rs232snd 98(), sbready 99()

11.10.5 rbready

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 gets available.

note buffer space for this function is always 1 character for VC/RT 5.0x for reasons of compatibility.

see also rs232rcv 99(), rbempty 101()

11.10.6 rbempty

rbempty — **receive buffer empty/serial interface**

synopsis — int rbempty(void)

description — This function checks the receive buffer for the serial interface. If it is empty, the function returns the value -1; otherwise, it returns the number of characters in the buffer.

Calling this function is especially recommendable when a character is to be read from the serial interface but it is not certain if any characters were received. In this case, calling the function rs232rcv() might cause the system to "hang".

note — buffer space for this function is always 1 character for VC/RT 5.0x for reasons of compatibility.

Note: — **Do not use this function for new development, since it is included for compatibility only. Use the function rbready () instead.**

see also — **rs232rcv** (), **rbready** ()

11.10.7 setbaud

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
9600 baud  */
```

11.10.8 kbdrcv

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 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.

The receive buffer can hold a maximum of 63 characters.

Note: neither hardware nor software handshaking is used in this routine

11.10.9 kbready

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 99() will "hang" until a character gets available.

see also kbdrcv 102(), rbready 100()

11.11 Low Level EPROM Access Functions

getf8 103	low-level function for reading a byte/flash EPROM (macro)
getf16 103	low-level function for reading a 16-bit word/flash EPROM (macro)
getf32 104	low-level function for reading a 32-bit word/flash EPROM (macro)
flpgm 104	low-level function for writing a byte/flash EPROM (macro)
flpgm8 105	low-level function for writing a byte/flash eprom
flpgm16 106	low-level function for writing a word/flash eprom
flpgm32 107	low-level function for writing a long-word/flash eprom
erase 107	low-level function for erasing sectors/flash EPROM
bdma	copy flash eprom to DMEM via BDMA

11.11.1 getf8

getf8 — **low-level function for reading a byte/flash EPROM (macro)**

synopsis

int getf8(long addr)

description

The function getf8() reads a byte from the flash EPROM. addr is the address of the memory location to be read.
The function returns the byte as the return value (int). The MSB of the return value is always 0.
The flash EPROM can be written to and read from, independent of the sectors it is divided into.
In contrast to writing, the function **getf8**()can read all bytes of the flash EPROM, even the boot sector.

11.11.2 getf16

getf16 — **low-level function for reading a 16-bit word/flash EPROM (macro)**

synopsis int getf16(long addr)

description The function getf16() reads a 16-bit word from the flash EPROM, i.e. two bytes from adjacent memory locations.
addr is the address of the first memory location to be read. This function returns the read byte and the following one as a 16-bit word return value (int).
The flash EPROM uses linear addressing.
The flash EPROM can be written to and read from independent of the sectors it is divided into.
In contrast to writing, the function **getf16** ()can read all bytes of the flash EPROM, even the boot sector

11.11.3 getf32

getf32 **low-level function for reading a 32-bit word/flash EPROM (macro)**

synopsis long getf32(long addr)

description The function getf32() reads a 32-bit word from the flash EPROM i.e. four bytes from adjacent memory locations.
addr is the address of the first memory location to be read. This function returns the read byte and the following three as a 32-bit word return value.
The flash EPROM uses linear addressing, starting at address 0L.
Thus, the flash EPROM can be written to and read from independent of the sectors it is divided into.
In contrast to writing, the function getf32()can read all bytes of the flash EPROM, even the boot sector.

11.11.4 flpgm

flpgm **low-level function for writing a byte/flash EPROM (macro)**

synopsis int flpgm(long addr, int value)

description The function flpgm() writes (programs) a byte to the flash EPROM.
addr is the address of the memory location to be programmed, value is the value to be written to the flash EPROM.
The flash EPROM uses linear addressing, starting at address 0L.
The flash EPROM can be written to and read from independent of the sectors it is divided into.
Make sure to erase a memory location before writing (see the function erase 107).
Erasing resets a sector's bytes to 0xff.
A byte must first have the value 0xff before it can be overwritten.
The function flpgm() cannot write boot sector addresses, because the operating system is located there.
Writing a byte takes about 16 µsec.

This function returns the value 0 after the byte has been written.
If the byte could not be written, the function returns a negative value:

Error	Return value
Sector 0 accessed	-1
Memory location already programmed	-2

Note : Functions flpgm8 105() , flpgm16 106() and flpgm32 107() all use reversed arguments. It is therefore recommended not using flpgm() but flpgm8() instead.

11.11.5 flpgm8

flpgm8 **low-level function for writing a byte/flash EPROM**

synopsis int flpgm8(int value, long addr)

description

The function flpgm8() writes (programs) a byte to the flash EPROM.
addr is the address of the memory location to be programmed, value is the value to be written to the flash EPROM.
The flash EPROM uses linear addressing, starting at address 0L.
For the 2MB EPROMs the last memory location is 0x1fffffL .
Thus, the flash EPROM can be written to and read from independent of the sectors it is divided into.
Make sure to erase a memory location before writing (see the function erase 107).
Erasing resets a sector's bytes to 0xff.
A byte must first have the value 0xff before it can be overwritten.
The function flpgm8() cannot write boot sector addresses (0x000000L through 0x05ffffL), because the operating system is located there.
Writing a byte takes about 16 µsec.

This function returns the value 0 after the byte has been written.
If the byte could not be written, the function returns a negative value:

Error	Return value
Sector 0 accessed	-1
Memory location already programmed	-2

11.11.6 flpgm16

flpgm16 **low-level function for writing a word EPROM**

synopsis int flpgm16(int val, long addr)

description see flpgm8()

11.11.7 flpgm32

flpgm32 — **low-level function for writing a long-word to flash EPROM**

synopsis — int flpgm32(long val, long addr)

description — see flpgm8 105 ()

11.11.8 erase

erase — **low-level function for erasing sectors/flash EPROM**

synopsis — int erase(int sector)

description — This function erases a single sector of the flash EPROM.
The employed flash EPROMs have 2 MBytes and consist of 32 sectors of 64 KB each.
Sectors 0-5 are reserved for the operating system and some utilities, and thus cannot be erased.
Sectors 6 through 31 are available to the user and can be erased at any time.
Erasing a sector takes around 1.5 seconds (30 at most).

This function returns the value 0 after the sector is erased.
If the sector could not be erased (e.g., the user tried to erase sector 0), the function returns the value -1.

note — **This function exists only for campatibility reasons. Handle with greates care.**
If sectors are erased that are used by the directory structure, the camera will crash.

Use file based functions 57 **instead.**

11.12 Utility Functions

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

11.12.1 getvar

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 */
```

11.12.2 setvar

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    */
```

11.12.3 getlvar

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

11.12.4 setlvar

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.

11.12.5 getstptr

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.

11.12.6 getdp

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.

11.12.7 getbss

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.

11.13 Lookup Table Functions

Enter topic text here.

11.13.1 set_overlay_bit

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 Î [0,255]
bit Î [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 an 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       */
```

11.13.2 set_lut_comp

set_lut_comp **compatibility mode for earlier VC/RT versions**

synopsis void set_lut_comp(int r, int g, int b)

description Earlier versions of VC/RT have just one overlay bitplane. It was possible to use this overlay translucent, but in most cases it was assigned to one overlay color. If you have software compiled for earlier VC/RT versions you may use this function.
It assigns the color defined by r,g,b Î [0,255] to overlay bit 0. Since this is a translucent overlay plane only 2 translucent overlay planes remain if you choose this option.

example

```
set_lut_comp(255,255,0);      /*
yellow              */
set_ovlmask(1);   /* bit 0 active   */
```

11.13.3 set_translucent

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 Î [0,255]
table Î [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.      */
```

11.13.4 set_ovlmask

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 ¹ 0 to the mask registers with this function will activate the transfer by writing a 1 to OVLY_ACTIVE.

11.13.5 init_LUT

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.

11.14 Time Related Functions

c_time 114	convert system time -> extract time
c_date 114	convert system time -> extract date
c_timedate 115	convert system time -> extract date
ltime 115	convert system time -> extract local time (macro)
ldate 115	convert system time -> extract local date (macro)
ltimedate 116	convert system time -> extract local date and time (ma
gtime 116	convert system time -> extract GMT time (macro)
gdate 116	convert system time -> extract GMT date (macro)
gtimedate 117	convert system time -> extract GMT date and time (macro)
x_timedate 117	calculate system time
xtimedate 118	calculate system time and store in system variable SE (macro)

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 savings 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.

11.14.1 c_time

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 114(), c_timedate 115()

11.14.2 c_date

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 114(), c_timedate 115()

11.14.3 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 114(), c_date 114()

11.14.4 ltime

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 115(), gdate 116()

11.14.5 ldate

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 115(), gtime 116()

11.14.6 ltimedate

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 115(), ldate 115(), gtimedate 117()

11.14.7 gtime

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 116(), ltime 115()

11.14.8 gdate

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 115(), gtime 116()

11.14.9 gtimedate

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 116(), gdate 116(), ltimedate 116()

11.14.10 x_timedate

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 118()

11.14.11 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()

11.14.12 RTC_set_time

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+2
000);  //set internal clock
  setvar(LOWBAT,0); /* reset internal
lowbat */
  RTC_set_time(); /* program clock chip */
}
```

see also xtimedate 118()

11.15 TCP/IP Functions

Socket definition A socket is an abstraction that identifies an endpoint and includes:

type of socket ; one of:

datagram (uses UDP)
stream (uses TCP)

socket address identified by:

port number
IP address
It may have a **remote endpoint** .

Socket options Each socket has socket options, which define characteristics of the socket, such as:

checksum calculations

Ethernet-frame characteristics

IGMP membership

non-blocking (nowait options)

push operations

sizes of send and receive buffers

timeouts

11.15.1 Datagram Sockets

Connectionless A datagram socket is connectionless in that an application uses a socket without first establishing a connection. Therefore, an application specifies the destination address and destination port number for each data transfer. An application can prespecify a remote endpoint for a datagram socket if desired.

Unreliable transfer A datagram socket is used for datagram-based data transfer, which does not acknowledge the transfer. Because delivery is not guaranteed, a higher layer is responsible for ensuring that the data is acknowledged when necessary.

Block oriented A datagram socket is block oriented. This means that when an application sends a block of data, the bytes of data remain together. If an application writes a block of data of, say, 100

bytes, VC/RT sends the data to the destination in a single packet, and the destination receives 100 bytes of data.

11.15.2 Stream Sockets

Connection based A stream-socket connection is uniquely defined by an address-port number pair for each of the two endpoints in the connection. For example, a connection to a Telnet server uses the local IP address with a local port number, and the server's IP address with port number 23.

Reliable transfer A stream socket provides reliable, end-to-end data transfer. To use stream sockets, a client establishes a connection to a peer, transfers data, and then closes the connection. Barring physical disconnection, VC/RT guarantees that all sent data is received in sequence.

Character oriented A stream socket is character oriented. This means that VC/RT may split or merge bytes of data as it sends the data from one protocol stack to another. An application on a stream socket may perform, for example, two successive write operations of 100 bytes each, and VC/RT may send the data to the destination in a single packet. The destination may then receive the data using, for example, four successive read operations of 50 bytes each.

11.15.3 Comparison of Datagram and Stream Sockets

	Datagram socket	**Stream socket**
Protocol	UDP	TCP
Connection based	No	Yes
Reliable transfer	No	Yes
Transfer mode	Block	Character

11.15.4 Creating and using Sockets

An application follows the following general steps to create and use sockets. The steps are summarized in the following diagrams and described in subsequent sections.

1. **Create a new socket** by calling socket(), indicating whether the socket is a datagram socket or a stream socket.
2. **Bind the socket** to a local address by calling bind 133().
3. If the socket is a stream socket, **assign a remote IP address** by doing one of the following:

3a. calling connect 134()

3b. calling listen 139() followed by accept 131()

4. **Send data** by calling sendto 151() for a datagram socket or send 148() for a stream socket.
5. **Receive data** by calling recvfrom 142() for a datagram socket or recv() for a stream socket.
6. When data transfer is finished, optionally **destroy the socket** by calling shutdown 165().

11.15.5 Diagram: Creating and Using Datagram Sockets (UDP)

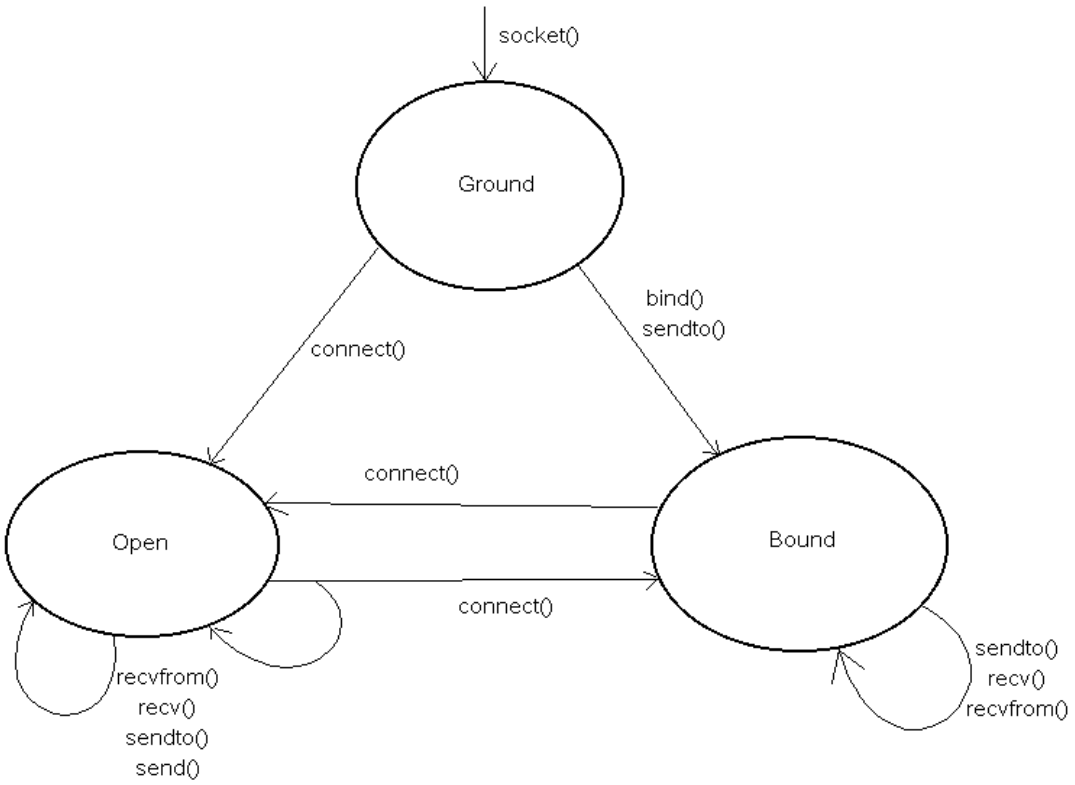


Diagram Creating and using datagram sockets (UDP)

11.15.6 Diagram: Creating and Using Stream Sockets (TCP)

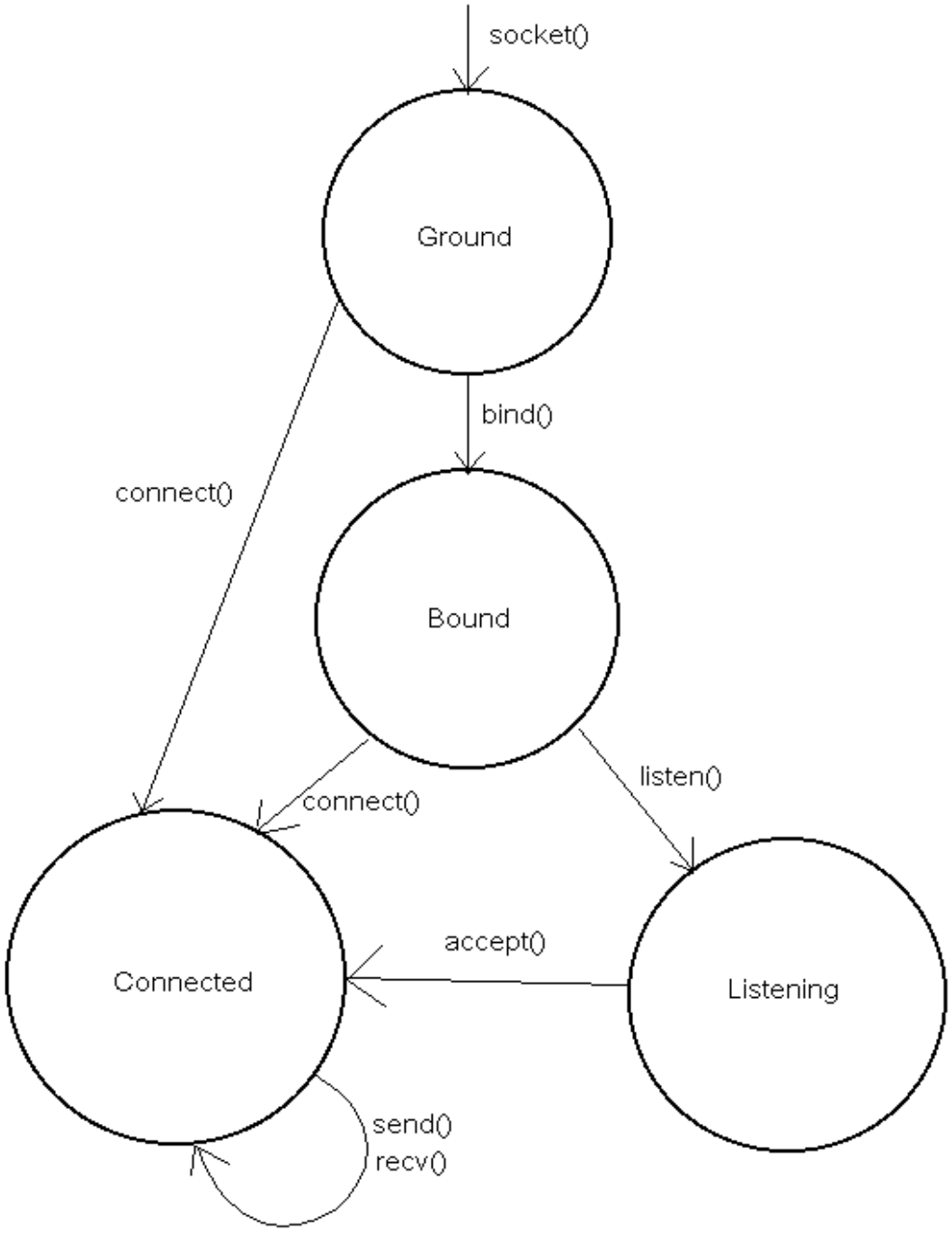


Diagram 2 Creating and using stream sockets (TCP)

11.15.7 Creating Sockets

To create a socket, an application calls socket() and specifies whether the socket is a datagram socket or a stream socket. The function returns a socket handle, which the application subsequently uses to access the socket.

11.15.8 Changing Socket Options

When VC/RT creates a socket, it sets all the socket options to **default values**.
To change the value of certain options, an application must do so **before it binds** the socket.

An application can change other options anytime.

All socket options and their default values are described in the following
setsockopt 154().

11.15.9 Binding Sockets

After an application creates a socket and optionally changes or sets socket
options, it must bind the socket to a local port number by calling bind(). The
function defines the endpoint of the local socket by the local IP address and port
number.

You can specify the local port number as any number, but if you specify zero,
VC/RT chooses an unused port number. To determine the port number that VC/RT
chose, call getsockname().
After the application binds the socket, how it uses the socket depends on whether
the socket is a datagram socket or a stream socket.

11.15.10 Using Datagram Sockets

11.15.10.1 Setting Datagram Socket Options

By default, VC/RT uses IGMP, and, by default, a socket is not in any group. The
application can change the following socket options for the socket:

- IGMP add membership
- IGMP drop membership
- send nowait
- checksum bypass

11.15.10.2 Transferring Datagram Data

An application transfers data by making calls to sendto 151() or send 148() and recvfrom 142() or recv 140().
With each call, VC/RT either sends or receives one UDP datagram, which contains up to 65,507 bytes of data.

If an application specifies more data, the functions return an error.

The functions send 148() and sendto 151() return when the data is passed to the Ethernet interface.

The functions recv 140() and recvfrom 142() return when the socket port receives the packet or immediately if a queued packet is already at the port. The receive buffer should be at least as large as the largest datagram that the application expects to receive. If a packet overruns the receive buffer, VC/RT truncates the packet and discards the truncated data.

11.15.10.3 Buffering

By default, send 148() and sendto 151() do not buffer outgoing data.

This behavior can be changed by using either the OPT_SEND_NOWAIT 163 socket option, or the VCRT_MSG_NONBLOCK send flag.

For incoming data, VC/RT matches the data, packet by packet, to recv 140() or recvfrom 142() calls that the application makes. If a packet arrives and a recv 140() or recvfrom 142() call is not waiting for data, VC/RT queues the packet.

11.15.10.4 Prescpecifying a peer

An application can optionally prespecify a peer by calling connect 134().

Prespecification has the following effect:

- send 148() can be used to send a datagram to the peer that is specified in the call to connect(). Calls to send 148() fail if connect() has not been called previously.
- the behavior of sendto 151() is unchanged. It is not restricted to the specified peer.
- the function recv 140() or recvfrom 142() returns datagrams that have been sent by the specified peer only

11.15.10.5 Shutting Down Datagram Sockets

An application can shut down a datagram socket by calling shutdown(). Before the function returns:

- outstanding calls to recvfrom() return immediately
- VC/RT discards received packets that are queued for the socket and frees their buffers

When shutdown() returns, the socket handle is invalid, and the application can no longer use the socket.

11.15.11 Using Stream Sockets

11.15.11.1 Changing Stream Socket Options

An application can change the value of certain stream-socket options anytime.

For details, see under setsockopt 154().

11.15.11.2 Establishing Stream Socket Connections

An application can establish a connection to a stream socket in one of these ways:

passively 128	by listening for incoming connection requests (by calling listen 139() followed by accept 131())
actively 128	by generating a connection request (by calling

connect())

11.15.11.2.1 Passive Establishing

By calling listen 139(), an application can passively put an unconnected socket in a listening state, after which the local socket endpoint responds to a single incoming connection request. After it calls listen 139(), the application calls accept 131(), which returns a new socket handle and lets the application accept the incoming connection request.

Usually,the application calls accept 131() immediately after it calls listen 139(). The application uses the new socket handle for all communication with the specified remote endpoint until one or both endpoints close the connection. The original socket remains in the listening state and continues to be referenced by the initial socket handle that socket() returned.

The new socket that the listen-accept mechanism creates inherits the socket options of the parent socket.

11.15.11.2.2 ActiveEstablishing

By calling connect(), an application can actively establish a stream-socket connection to the remote endpoint that the function specifies. If the remote endpoint is not in the listening state, connect() fails. Depending on the state of the remote endpoint, connect() fails immediately or after the time that the connect-timeout socket option specifies.
If the remote endpoint accepts the connection, the application uses the original socket handle for all its communication with that remote endpoint, and VC/RT maintains the connection until either or both endpoints close the connection.

11.15.11.3 Getting Stream Socket Names

After an application establishes a stream-socket connection, it can get the identifiers for the local endpoint (by calling getsockname()) and for the remote endpoint (by calling getpeername()).

11.15.11.4 Sending Stream Data

An application sends data on a stream socket by calling send(). When the function returns depends on the values of the send nowait (OPT_SEND_NOWAIT) socket option. An application can change the value by calling setsockopt().

11.15.11.4.1 send nowait (nonblocking I/O)

send() returns **FALSE** (default) when TCP has buffered all data but not necessarily sent it
send() returns **TRUE** Immediately (the result is a filled or partially filled buffer)

11.15.11.5 Receiving Stream Data

An application receives data on a stream socket by calling recv(). The application passes the function a buffer, into which VC/RT places the incoming data. When the function returns depends on the values of the receive-nowait (OPT_RECEIVE_NOWAIT) and receive-push (OPT_RECEIVE_PUSH) socket options. The application can change the values by calling setsockopt().

Receive nowait (non-blocking I/O)	**Receive push (delay transmission)**	***recv*() returns when:**
FALSE (default)	TRUE (default)	One of: •a push flag in the data is received •supplied buffer is completely filled with incoming data •receive timeout expires (the default receive timeout is an unlimited time)
FALSE (default)	FALSE	Either: • supplied buffer is completely filled with incoming data •receive timeout expires
TRUE	(ignored)	Immediately after it polls TCP for any data in the internal receive buffer

11.15.11.6 Buffering Data

The size of the VC/RT per-socket send buffer is determined by the socket option that controls the size of the send buffer. VC/RT copies data into its send buffer from the buffer that the application supplies. As the peer acknowledges the data, VC/RT releases space in its buffer. If the buffer is full, calls to send() with the send-push (OPT_SEND_PUSH 163) socket option FALSE block until the remote endpoint acknowledges some or all the data.
The size of the VC/RT per-socket receive buffer is determined by the socket option that controls the size of the receive buffer. VC/RT uses the buffer to hold incoming data when there are no outstanding calls to recv(). When the application calls recv(), VC/RT copies data from its buffer to the buffer that the application supplies, and, consequently, the remote endpoint can send more

data.

11.15.11.7 Improving the Throughput of Stream Data

- Include the push flag in sent data only where the flag is needed; that is, at the end of a stream of data.
- Specify the largest possible send and receive buffers to reduce the amount of work that the application and VC/RT do.
- When you call recv(), call it again immediately to reduce the amount of data that VC/RT must copy into its receive buffer.
- Specify the size of the send and receive buffers to be multiples of the maximum packet size.
- Call send 148() with an amount of data that is a multiple of the maximum packet size.

11.15.11.8 Shutting Down Stream Sockets

11.15.11.8.1 Shutting Down Gracefully

If the socket is to be shut down gracefully, VC/RT tries to deliver all the data that is in its send buffer for the socket. As specified by the TCP specification, VC/RT maintains the socket connection for four minutes after the remote endpoint disconnects.

11.15.11.8.2 Shutting Down with an abort operation

If the socket is to be shut down with an abort operation:

- VC/RT immediately discards the socket and the socket's internal send and receive buffers.
- The remote endpoint frees its socket immediately after it sends all the data that is in its send buffer.

11.15.11 Summary of Socket Functions

accept	Accept the next incoming stream connection and clone the socket to create a new socket, which services the connection
bind	Identify the local application endpoint by providing a port Number
connect	Establish a stream connection with an application endpoint or set a remote endpoint for a datagram socket
getpeername	Determine the peer address-port number endpoint of a connected socket
getsockname	Determine the local address-port number endpoint of a bound socket
getsockopt	Get the value of a socket option
listen	Allow incoming stream connections to be received on the port number that is identified by a socket
recv	Receive data on a stream or datagram socket
recvfrom	Receive data on a datagram socket
VCRT_attachsock	Get access to a socket that is owned by another task
VCRT_detachsock	Relinquish ownership of a socket
VCRT_geterror	Get the reason why an VC/RT function returned an error for the socket
VCRT_selectall	Wait for activity on any socket that a caller owns
VCRT_selectset	Wait for activity on any socket in a set of sockets
send	Send data on a stream socket or on a datagram socket for which a remote endpoint has been specified
sendto	Send data on a datagram socket
setsockopt	Set the value of a socket option
shutdown	Shutdown a connection and discard the socket
Socket_stream	Create a stream socket
Socket_dgram	Create a datagram socket

11.15.12 accept

accept — **create a new stream socket to accept incoming connections**

synopsis — uint_32 accept(uint_32 socket, sockaddr_in *peeraddr, uint_16 *addrlen)

parameters

socket [IN]	Handle for the parent stream socket
peeraddr [OUT]	Pointer to where to place the remote endpoint identifier
addrlen [IN/OUT]	[IN] Pointer to the length, in bytes,of what *peeraddr* points to
	[OUT] Full size, in bytes, of the remote-endpoint identifier

returns Handle for a new stream socket
VCRT_SOCKET_ERROR

traits Blocks until an incoming connection is available

see also bind(), connect(), listen(), socket()

description The function accepts incoming connections by creating a new stream socket for the connections. The parent socket (socket) must be in the listening state; it remains in the listening state after each new socket is created from it.
The new socket has the same local endpoint and socket options as the parent; the remote endpoint is the originator of the connection.

example

```
uint_32 handle;
uint_32 child_handle;
sockaddr_in remote_sin;
uint_16 remote_addrlen;
uint_32 status;
...
status = listen(handle, 0);
if (status != VCRT_OK) {
printf("\nError, listen() failed with error code %lx", status);
} else {
remote_addrlen = sizeof(remote_sin);
child_handle = accept(handle, &remote_sin, &remote_addrlen);
if (child_handle != VCRT_SOCKET_ERROR) {
printf("\nConnection accepted from %lx, port %d",
remote_sin.sin_addr, remote_sin.sin_port);
} else {
status = VCRT_geterror(handle);
if (status == VCRT_OK) {
printf("\nConnection reset by peer");
```

```
} else {
printf("Error, accept() failed with error code %lx",
status);
}
}
}
```

11.15.12. bind

bind **bind the local address to the socket**

synopsis uint_32 bind(uint_32 socket, sockaddr_in *localaddr, uint_16 addrlen)

parameters

socket [IN]	Socket handle for the socket to bind
localaddr [IN]	Pointer to the local endpoint identifier to which to bind socket (see description)
addrlen [IN]	Length in bytes of what localaddr points to

returns VCRT _OK
Specific error code

traits Blocks, but VC/RT immediately services the command and is replied to by the socket layer

see also socket()

description

Field in sockaddr_in:	Must have this input value:
sin_family	AF_INET
sin_port	One of: •local port number for the socket •0 (To determine the port number that VC/RT chooses, call getsockname())
sin_addr	One of: •IP address that was previously bound •INADDR_ANY

Usually, TCP/IP servers bind to INADDR_ANY, so that one instance of the server can service

all IP addresses.

example: Bind a socket to port number 2010.

```
uint_32 sock;
sockaddr_in local_sin;
uint_32 result;
...
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock == VCRT_SOCKET_ERROR)
{
printf("\nError, socket create failed");
return;
}
memset((char *) &local_sin, 0, sizeof(local_sin));
local_sin.sin_family = AF_INET;
local_sin.sin_port = 2010;
local_sin.sin_addr.s_addr = INADDR_ANY;
result = bind(sock, &local_sin, sizeof (sockaddr_in));
if (status != VCRT_OK)
printf("\nError, bind() failed with error code %lx",
                result);
```

11.15.12.connect

connect — **Connect the stream socket to the remote endpoint**

synopsis — uint_32 connect(uint_32 socket, sockaddr_in *destaddr, uint_16 addrlen)

parameters

socket [IN]	Handle for the stream socket to connect
destaddr [IN]	Pointer to the remote endpoint identifier
addrlen [IN]	Length in bytes of what destaddr points to

returns — VCRT _OK (success)
Specific error code (failure)

traits — Blocks until the connection is accepted or until the connection-timeout socket option expires

see also — accept 131(), bind 133(), getsockopt 138(), listen 139(), setsockopt 154(), socket_dgram 167(), socket_stream 167()

description — **Stream socket** :
The function fails if the remote endpoint rejects

the connection request, which it may do immediately is unreachable, which causes the connection timeout to expire
If the function is successful, the application can use the socket to transfer data.

Datagram socket:
The function connect() has the following effects on a datagram socket:
send() can be used instead of sendto() to send a datagram to destaddr the behavior of sendto() is unchanged: it can still be used to send a datagramto any peer the socket receives datagrams from destaddr only
connect () may be used multiple times.
Whenever **connect** () is called, the current endpoint is replaced by the new one.
A connection can be dissolved by calling connect() and specifying an address family of AF_UNSPEC. This dissolves the association, places the socket in the bound state, and returns the error code VCRTERR_SOCK_INVALID_AF.
Should connect() fail, the socket will be in a bound state (no remote endpoint).

example: Stream socket

```
uint_32 sock;
uint_32 child_handle;
sockaddr_in remote_sin;
uint_16 remote_addrlen = sizeof(sockaddr_in);
uint_32 result;
...
/* Connect to 192.203.0.83, port 2011: */
memset((char *) &remote_sin, 0, sizeof(sockaddr_in));
remote_sin.sin_family = AF_INET;
remote_sin.sin_port = 2011;
remote_sin.sin_addr.s_addr = 0xC0A80001; /* 192.168.0.1 */
result = connect(sock, &remote_sin, remote_addrlen);
if (result != VCRT_OK)
{
printf("\nError--connect() failed with error code %lx.",
result);
} else {
printf("\nConnected to %lx, port %d.",
remote_sin.sin_addr.s_addr, remote_sin.sin_port);
}
```

11.15.12.4 ENET_get_stats

ENET_get_stats — **get a pointer to the Ethernet statistics that VCRT collects**

synopsis — ENET_STATS *ENET_get_stats(enet_handle *handle)

parameters — handle [IN] Pointer to the Ethernet handle

returns — Pointer to an ENET_STATS structure

traits

see also — ICMP_stats(), IP_stats(), IPIF_stats(), TCP_stats(), UDP_stats(), ENET_STATS

description

example

```
ENET_STATS *enet;
_enet_handle ehandle;
...
enet = ENET_get_stats();
printf("\n%d Ethernet packets received", enet->ST_RX_TOTAL);
```

11.15.12.5 getpeername

getpeername — **get the remote-endpoint identifier of a socket**

synopsis — uint_32 getpeername(uint_32 socket, sockaddr_in *name, uint_16 *namelen)

Parameters

socket [IN]	Handle for the stream socket
name [OUT]	Pointer to a placeholder for the remote-endpoint identifier of the socket
namelen [IN/OUT]	[IN] Pointer to the length, in bytes, of what name points to
	[OUT] Full size, in bytes, of the remote-endpoint identifier

returns — VCRT_OK (success)
Specific error code (failure)

traits — Blocks, but the command is immediately serviced and replied to

see also accept 131(), connect 134(), getsockname 137(), socket()

description The function returns the remote endpoint for the socket as was determined by connect 134() or accept 131().

example

```
uint_32 handle;
sockaddr_in remote_sin;
uint_32 status;
uint_16 namelen;
...
namelen = sizeof (sockaddr_in);
status = getpeername(handle, &remote_sin, &namelen); if (status !=
VCRT_OK)
{
printf("\nError, getpeername() failed with error code %lx",
status);
} else {
printf("\nRemote address family is %x", remote_sin.sin_family);
printf("\nRemote port is %d", remote_sin.sin_port);
printf("\nRemote IP address is %lx",
remote_sin.sin_addr.s_addr);
}
```

11.15.12. getsockname

getsockname **Get the local-endpoint identifier of the socket**

synopsis uint_32 getsockname(uint_32 socket,sockaddr_in *name, uint_16 *namelen)

parameters

socket [IN]	Socket handle
name [OUT]	Pointer to a placeholder for the local-endpoint identifier of the socket
namelen [IN/OUT]	[IN] Pointer to the length, in bytes, of what name points to
	[OUT] Full size, in bytes, of the remote-endpoint identifier

returns VCRT_OK (success)
Specific error code (failure)

traits Blocks, but the command is immediately serviced and replied to

see also bind 133(), getpeername 136(), socket()

description The function returns the local endpoint for the socket as was defined by bind 133().

example

```
uint_32 handle;
sockaddr_in local_sin;
uint_32 status;
uint_16 namelen;
...
namelen = sizeof (sockaddr_in);
status = getsockname(handle, &local_sin, &namelen);
if (status != VCRT_OK)
{
printf("\nError, getsockname() failed with error code %lx",
status);
} else {
printf("\nLocal address family is %x", local_sin.sin_family);
printf("\nLocal port is %d", local_sin.sin_port);
printf("\nLocal IP address is %lx", local_sin.sin_addr.s_addr);
}
```

11.15.12. getsockopt

getsockopt **Get the value of the socket option**

synopsis uint_32 getsockopt(uint_32 socket, int_32 level, uint_32 optname, pointer optval, uint_32 *optlen)

parameters

socket [IN]	Socket handle
level [IN]	Protocol level at which the option resides
optname [IN]	Option name (see description)
optval [IN/OUT]	Pointer to the option value
optlen [IN/OUT]	[IN] Size of optval in bytes
	[OUT] Full size, in bytes, of the option value

returns VCRT_OK (success)
Specific error code (failure)

traits Blocks, but the command is immediately serviced and replied to

see also setsockopt 154()

description An application can get all socket options for all protocol levels. For a complete description of socket options and protocol levels, see setsockopt 154().

11.15.12.4 listen

listen **put the stream socket into the listening state**

synopsis uint_32 listen(uint_32 socket, uint_16 backlog)

parameters

socket [IN]	Socket handle
backlog [IN]	Ignored

returns VCRT_OK (success)
Specific error code (failure)

traits Blocks, but the command is immediately serviced and replied to

see also accept 131(), bind 133(), socket()

description After the application calls listen(), it should call accept() to attach new sockets to the incoming requests.

example See accept 131().

11.15.12.5 VCRT_ping

VCRT_ping **send an ICMP echo-request packet to the IP address and wait for a reply**

synopsis uint_32 VCRT_ping(ip_address address, uint_32 *timeout, uint_16 id)

parameters

address [IN]	IP address to which to send the packet
timeout [IN/OUT]	[IN] One of:
	Pointer to the maximum time to wait for a reply
	0 (wait indefinitely)
	[OUT] Pointer to the round-trip time
id [IN]	User ID for the echo request

returns

VCRT_OK (success)
Error code (failure)

11.15.12.recv

recv

provide VCRT with the buffer in which to place incoming stream data

synopsis

int_32 recv(uint_32 socket, char *buffer, uint_32 buflen, uint_32 flags)

parameters

socket [IN]	Handle for the connected stream socket
buffer [OUT]	Pointer to the buffer in which to place received data
buflen [IN]	Size of buffer in bytes
flags [IN]	Flags to underlying protocols. One of: VCRT_MSG_PEEK - For a UDP socket, receive a datagram but don't consume it (ignored for stream sockets) 0 - Ignore

returns

Number of bytes received (success)
VCRT_ERROR (failure)

traits

May block, but the command is immediately serviced

If non-blocking I/O is **disabled** on the socket, the function blocks until data satisfying the receive-push socket option is received

If non-blocking I/O is **enabled** on the socket, the command is immediately replied to, returning whatever incoming data is buffered internally

see also

accept 131(), bind 133(), getsockopt 138(), listen 139(), VCRT_geterror 145(), send 148(), setsockopt 154(), shutdown 165(), socket()

description

When the flags parameter is VCRT_MSG_PEEK, the same datagram is received the next time recv() or recvfrom() is called.

If the function returns VCRT_ERROR, the application can call VCRT_geterror() to determine the reason for the error.

Stream socket

If the receive-nowait socket option is TRUE, VCRT immediately copies internally buffered data (up to buflen bytes) into the buffer (at buffer), and recv() returns. If the receive-wait socket option is TRUE, recv() blocks until the buffer is full or the receive-push socket option is satisfied.

If the receive-push socket option is TRUE, a received TCP push flag causes recv() to return with whatever data has been received. If the receive-push socket option is FALSE, VCRT ignores incoming TCP push flags, and recv() returns when enough data has been received to fill the buffer.

Datagram socket

The recv() function on a datagram socket is identical to recvfrom() with NULL fromaddr and fromlen pointers. The recv() function is normally used on a connected socket.

example: Stream socket

```
uint_32 handle;
char buffer[20000];
uint_32 count;
...
count = recv(handle, buffer, 20000, 0);
if (count == VCRT_ERROR)
```

```
{
printf("\nError, recv() failed with error code %lx",
VCRT_geterror(handle));
} else {
printf("\nReceived %ld bytes of data.", count);
}
```

11.15.12.: recvfrom

recvfrom — **provide VC/RT with the buffer in which to place incoming datagram socket data**

synopsis — int_32 recvfrom(uint_32 socket, char *buffer, uint_32 buflen, uint_32 flags, sockaddr_in *fromaddr, uint_16 *fromlen)

parameters

socket [IN]	Handle for the datagram socket
buffer [OUT]	Pointer to buffer in which to place received data
buflen [IN]	Number of bytes in the buffer
flags [IN]	Flags to underlying protocols. One of: VCRT_MSG_PEEK - Receive a datagram but don't consume it 0 - Ignore
fromaddr [OUT]	Source socket address of the message
fromlen [IN/OUT]	[IN] Size of the fromaddr buffer [OUT] Size of the socket-address stored in the fromaddr buffer, or if the provided buffer was too small (socket-address was truncated), the length before truncation

returns — Number of bytes received (success)
VCRT_ERROR (failure)

traits — Blocks until data is available or an error occurs

see also — bind 133(), VCRT_geterror 145(), sendto 151(), socket()

description If a remote endpoint has been specified with connect(), only datagrams from that source will be received.
When the flags parameter is VCRT_MSG_PEEK, the same datagram is received the next time recv() or recvfrom() is called.
If fromlen is NULL, the socket address is not written to fromaddr. If fromaddr is NULL and the value of fromlen is not NULL, the result is unspecified.
If the function returns VCRT_ERROR, the application can call VCRT_geterror 145() to determine the reason for the error.

example: **Receive up to 500 bytes of data.**

```
uint_32 handle;
sockaddr_in remote_sin;
uint_32 count;
char my_buffer[500];
uint_16 remote_len = sizeof(remote_sin);
...
count = recvfrom(handle, my_buffer, 500, 0,
&remote_sin, &remote_len);
if (count == VCRT_ERROR)
{
printf("\nrecvfrom() failed with error %lx",
VCRT_geterror(handle));
} else {
printf("\nReceived %ld bytes of data.", count);
}
```

11.15.12. VCRT_attachsock

VCRT_attachsock **take ownership of the socket**

synopsis uint_32 VCRT_attachsock(uint_32 socket)

parameters socket [IN] Socket handle

returns new socket handle (success)
VCRT_SOCKET_ERROR (failure)

traits Blocks, although the command is serviced and responded to immediately

see also accept 131(), VCRT_detachsock 144()

description The function adds the calling task to the

socket's list of owners.

example

A main task loops to accept connections. When it accepts a connection, it creates a child task to manage the connection: it relinquishes control of the socket by calling VCRT_detachsock() and then creates the child with the accepted socket handle as the initial parameter.

```
while (TRUE)
{
  /* Issue ACCEPT: */
  TELNET_accept_skt =
  accept(TELNET_listen_skt, &peer_addr,
&addr_len);
  if (TELNET_accept_skt !=
VCRT_SOCKET_ERROR)
  {
    /* Transfer the socket and create the
child task to look after the     socket:
*/
    if (VCRT_detachsock(TELNET_accept_skt)
== VCRT_OK)
    {
      child_task =
(_task_create(LOCAL_ID,
CHILD),TELNET_accept_skt);
    }
    else
    {
      printf("\naccept() failed, error
      0x%lx",
VCRT_geterror(TELNET_accept_skt));
    }
  }
}
```

11.15.12. VCRT_detachsock

VCRT_detachsock **relinquish ownership of the socket**

synopsis uint_32 VCRT_detachsock(uint_32 socket)

parameters socket [IN] Socket handle from socket(), accept 131(), or VCRT_attachsock 143()

returns VCRT_OK (success)
Specific error code (failure)

traits Blocks, although the command is serviced and responded to immediately

see also accept 131(), VCRT_attachsock 143(), socket()

description The function removes the calling task from the socket™s list of owners.

example See VCRT_attachsock 143().

11.15.12.: VCRT_geterror

VCRT_geterror **Get the reason why an VC/RT function returned an error for the socket**

synopsis uint_32 VCRT_geterror(uint_32 socket)

parameters socket [IN] Socket handle

returns VCRT_OK (no socket error)
Last error code for the socket

traits Does not block

see also accept(), recv(), recvfrom(), send(), sendto()

description Use the function if accept() returns VCRT_SOCKET_ERROR or any of the following functions returns VCRT_ERROR:

recv 140()
recvfrom 142()
send 148()
sendto 151()

example See accept 131(), recv 140(), recvfrom 142(), send 148(), and sendto 151().

11.15.12.: VCRT_selectall

VCRT_selectall **wait for activity on any socket that the caller owns**

synopsis uint_32 VCRT_selectall(uint_32 timeout)

parameters

timeout [IN]	One of: Maximum number of milliseconds to wait for activity 0 (wait indefinitely) -1 (do not block)

returns

Socket handle (activity was detected; see description)
0 (timeout expired)
VCRT_SOCKET_ERROR (error)

traits

If timeout is not -1, blocks until activity is detected on any socket that the calling task owns

see also

VCRT_selectset()

description

Activity consists of any of the following.

This type of socket:	Receives:
Unbound datagram	Datagrams
Listening stream	Connection requests
Connected stream	Data or Shutdown requests that are initiated by the remote Endpoint

Example **Echo data on TCP port number 7.**

```
int_32 servsock;
int_32 connsock;
int_32 status;
SOCKET_ADDRESS_STRUCT addrpeer;
uint_16 addrlen;
char buf[500];
int_32 count;
uint_32 error
/* create a stream socket and bind it to port 7: */
error = listen(servsock, 0);
if (error != VCRT_OK) {
printf("\nlisten() failed, status = %d", error);
return;
}
for (;;) {
connsock = VCRT_selectall(0);
if (connsock == VCRT_SOCKET_ERROR) {
printf("\nVCRT_selectall() failed!");
} else if (connsock == servsock) {
status = accept(servsock, &addrpeer, &addrlen);
if (status == VCRT_SOCKET_ERROR)
```

```
printf("\naccept() failed!");
} else {
count = recv(connsock, buf, 500, 0);
if (count <= 0)
shutdown(connsock, FLAG_CLOSE_TX);
else
send(connsock, buf, count, 0);
}
}
```

11.15.12. VCRT_selectset

VCRT_selectset — **wait for activity on any socket in the set of sockets**

synopsis

uint_32 VCRT_selectset(pointer sockset, uint_32 count, uint_32 timeout)

parameters

sockset [IN]	Pointer to an array of sockets
count [IN]	Number of sockets in the array
timeout [IN]	One of: Maximum number of milliseconds to wait for activity 0 (wait indefinitely) -1 (do not block)

returns

Socket handle (activity was detected)
0 (timeout expired)
VCRT_SOCKET_ERROR (error)

traits

If timeout is not -1, blocks until activity is detected on at least one of the sockets in the set

see also

VCRT_selectall 145 ()

description

For the description of what constitutes activity, see VCRT_selectall().

Example — **Echo UDP data that is received on ports 2010, 2011, and 2012.**

```
int_32 socklist[3];
sockaddr_in local_sin;
uint_32 result;
...
memset((char *) &local_sin, 0, sizeof(local_sin));
local_sin.sin_family = AF_INET;
```

```
local_sin.sin_addr.s_addr = INADDR_ANY;
local_sin.sin_port = 2010;
socklist[0] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[0], &local_sin, sizeof
                    (sockaddr_in));
local_sin.sin_port = 2011;
socklist[1] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[1], &local_sin, sizeof
                    (sockaddr_in));
local_sin.sin_port = 2012;
socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[2], &local_sin, sizeof
                    (sockaddr_in));
while (TRUE) {
sock = VCRT_selectset(socklist, 3, 0);
rlen = sizeof(raddr);
length = recvfrom(sock, buffer, BUFFER_SIZE, 0, &raddr,
                    &rlen);
sendto(sock, buffer, length, 0, &raddr, rlen);
}
```

11.15.12. send

send — **Send data on the stream socket, or on a datagram socket for which a remote endpoint has been specified.**

synopsis — int_32 send(uint_32 socket, char *buffer, uint_32 buflen, uint_32 flags)

parameters

socket [IN]	Handle for the socket on which to send data	
buffer [IN]	Pointer to the buffer of data to send	
buflen [IN]	Number of bytes in the buffer (no restriction)	
flags [IN]	Flags to underlying protocols, selected from three independent groups. Perform a bitwise OR of one flag only from one or more of the following groups: **Group 1:**	
	VCRT_MSG_BLOCK	Override the OPT_SEND_NOWAIT datagram socket option; make it behave as though it is FALSE
	VCRT_MSG_NONBLOCK	Override the OPT_SEND_NOWAIT datagram socket option; make it behave as though it is TRUE
	Group 2:	
	VCRT_MSG_CHKSUM	Override the OPT_CHECKSUM_BYPASS checksum bypass option; make it behave as though it is FALSE
	VCRT_MSG_NOCHKSUM	Override the OPT_CHECKSUM_BYPASS checksum bypass option; make it behave as though it is TRUE
	Group 3:	
	VCRT_MSG_NOLOOP	Do not send the datagram to the loopback interface
	0	Ignore

returns Number of bytes sent (success)
VCRT_ERROR (failure)

traits May block until data is placed in the socket's send buffer, whose size is set by setsockopt()

see also accept 131(), bind 133(), getsockopt 138(),

listen 139(), recv 140(), VCRT_geterror 145(), setsockopt 154(), shutdown 165(), socket()

description

If the function returns VCRT_ERROR, the application can call VCRT_geterror 145() to determine the cause of the error.

Stream socket
VC/RT packetizes the data (at buffer) into TCP packets and delivers the packets reliably and sequentially to the connected remote endpoint.

If the send-nowait socket option is TRUE, VC/RT immediately copies the data into the internal send buffer for the socket, to a maximum of buflen. The function then returns.

If the send-push socket option is TRUE, VC/RT appends a push flag to the last packet that it uses to send the buffer; all data is sent immediately, taking into account the capabilities of the remote endpoint buffer.

Datagram socket If a remote endpoint has been specified using connect(),

send 148() is identical to sendto 151() using the specified remote endpoint. If a remote endpoint has not been specified, send 148() returns VCRT_ERROR.

The flags parameter can be used for datagram sockets only. The override is temporary and lasts for the current call to send 148() only.

Setting flags to VCRT_MSG_NOLOOP is useful when broadcasting or multicasting a datagram to several destinations. When flags is set to VCRT_MSG_NOLOOP, the datagram is not duplicated for the local host interface.

example: Stream socket

```
uint_32 handle;
char buffer[20000];
uint_32 count;
...
count = send(handle, buffer, 20000, 0);
```

```
if (count == VCRT_ERROR)
  printf("\nError, send() failed with error code %lx",
VCRT_geterror(handle));
```

11.15.12. sendto

sendto	**send data on the datagram socket**
synopsis	int_32 sendto(uint_32 socket, char *buffer, uint_32 buflen, uint_16 flags, sockaddr_in *destaddr, uint_16 addrlen)
parameters	

socket [IN]	Handle for the socket on which to send data	
buffer [IN]	Pointer to the buffer of data to send	
buflen [IN]	Number of bytes in the buffer	
flags [IN]	Flags to underlying protocols, selected from three independent groups. Perform a bitwise OR of one flag only from one or more of the ollowing groups: **Group 1:**	
	VCRT_MSG_BLOCK	Override the OPT_SEND_NOWAIT datagram socket option; make it behave as though it is FALSE
	VCRT_MSG_NONBLOCK	Override the OPT_SEND_NOWAIT datagram socket option; make it behave as though it is TRUE
	Group 2:	
	VCRT_MSG_CHKSUM	Override the OPT_CHECKSUM_BYPASS checksum bypass option; make it behave as though it is FALSE
	VCRT_MSG_NOCHKSUM	Override the OPT_CHECKSUM_BYPASS checksum bypass option; make it behave as though it is TRUE
	Group 3:	
	VCRT_MSG_NOLOOP	Do not send the datagram to the loopback interface
	0	Ignore

destaddr [IN] Remote endpoint identifier to which to send the data

addrlen [IN] Number of bytes pointed to by destaddr

returns Number of bytes sent (success)
VCRT_ERROR (failure)

traits Blocks, but the command is immediately serviced and replied to

see also setsockopt 154(), bind 133(), recvfrom 142(), VCRT_geterror 145(), socket()

description The function sends the data (at buffer) as a UDP datagram to the remote endpoint (at destaddr).

This function can also be used when a remote endpoint has been prespecified through connect(). The datagram is sent to destaddr even if it is different than the prespecified remote endpoint.

If the socket address has been prespecified, you can call sendto() with destaddr set to NULL and addrlen equal to zero: this combination sends to the prespecified address. Calling sendto() with destaddr set to NULL and addrlen equal to zero without first having prespecified the destination will result in an error.

The flags parameter can be used for datagram sockets only. The override is temporary and lasts for the current call to sendto() only. Setting flags to VCRT_MSG_NOLOOP is useful when broadcasting or multicasting a datagram to several destinations. When flags is set to VCRT_MSG_NOLOOP, the datagram is not duplicated for the local host interface.

If the function returns VCRT_ERROR, the application can call VCRT_geterror 145() to determine the cause of the error.

Example **Send 500 bytes of data to IP address 192.203.0.54, port number 678.**

```
uint_32 handle;
sockaddr_in remote_sin;
uint_32 count;
char my_buffer[500];
...
for (i=0; i < 500; i++) my_buffer[i]= (i & 0xff);
memset((char *) &remote_sin, 0, sizeof(sockaddr_in));
remote_sin.sin_family = AF_INET;
```

```
remote_sin.sin_port = 678;
remote_sin.sin_addr.s_addr = 0xC0CB0036;
count = sendto(handle, my_buffer, 500, 0, &remote_sin,
sizeof(sockaddr_in));
if (count != 500)
printf("\nsendto() failed with count %ld and error %lx",
count, VCRT_geterror(handle));
```

11.15.12. setsockopt

setsockopt — **set the value of the socket option**

synopsis — uint_32 setsockopt(uint_32 socket, uint_32 level, uint_32 optname, pointer optval, uint_32 optlen)

parameters

socket [IN]	One of: If level is anything but SOL_NAT, handle for the socket whose option is to be changed If level is SOL_NAT, socket is ignored
level [IN]	Protocol level at which the option resides; one of: SOL_IGMP SOL_LINK SOL_NAT (not available) SOL_SOCKET SOL_TCP SOL_UDP
optname [IN]	Option name; see description
optval [IN]	Pointer to the option value
optlen [IN]	Number of bytes that optval points to

returns — VCRT_OK (success)
Specific error code (failure)

traits — Blocks, but the command is immediately serviced and replied to

see also — bind 133(), getsockopt 138(), ip_mreq, nat_ports, nat_timeouts

Description — You can set most socket options by calling setsockopt(). However, the following options cannot be set; you can use them only with getsockopt 138():

IGMP get membership
receive Ethernet 802.1Q priority tags
receive Ethernet 802.3 frames
socket error
socket type

Settable options have default values. If you want to change the value of some settable options, you must do so before you bind the socket.
For other settable options, you can change the value anytime after the socket is created.

NOTE Some options can be temporarily overridden for datagram sockets. For more information, see send 148() and sendto 151().

11.15.12.19.1 Option Names

11.15.12.19.1.1 OPT_CHECKSUM_BYPASS

Checksum bypass

Option name	OPT_CHECKSUM_BYPASS (can be overridden)
Protocol level	SOL_UDP
Values	TRUE VC/RT sets to 0 the checksum field of sent datagram packets, and the generation of checksums is bypassed FALSE VC/RT generates checksums for sent datagram packets
Default value	FALSE
Change	Before bound
Socket type	Datagram
Comments	

11.15.12.19.1.2 OPT_CONNECT_TIMEOUT

Connect timeout

Option name	OPT_CONNECT_TIMEOUT
Protocol level	SOL_TCP
Values	>= 180,000 VC/RT maintains the connection

	for this number of milliseconds
Default value	480,000 (8 min)
Change	Before bound
Socket type	Stream
Comments	Connect timeout corresponds to R2 (as defined in RFC 793) and is sometimes called the hard timeout. It indicates how much time VC/RT spends attempting to establish a connection before it gives up. If the remote endpoint does not acknowledge a sent segment within the connect timeout (as would happen if a cable breaks, for example), VC/RT shuts down the socket connection, and all function calls that use the connection return.

11.15.12.19.1.3 VCRT_SO_IGMP_ADD_MEMBERSHIP

IGMP add membership

Option name	VCRT_SO_IGMP_ADD_MEMBERSHIP
Protocol level	SOL_IGMP
Values	
Default value	Not in a group
Change	Anytime
Socket type	Datagram
Comments	IGMP must be in the VC/RT protocol table.

Example

To join a multicast group:

```
uint_32 sock;
struct ip_mreq group;
group.imr_multiaddr =
multicast_ip_address;
group.imr_interface = local_ip_address;
error = setsockopt(sock, SOL_IGMP,
VCRT_SO_IGMP_ADD_MEMBERSHIP,
&group,sizeof(group));
```

11.15.12.19.1.4 VCRT_SO_IGMP_DROP_MEMBERSHIP

IGMP drop membership

Option name	VCRT_SO_IGMP_DROP_MEMBERSHIP
Protocol level	SOL_IGMP
Values	
Default value	Not in a group
Change	After the socket is created
Socket type	Datagram
Comments	IGMP must be in the VC/RT protocol table.

Example

To leave a multicast group:

```
uint_32 sock;
struct ip_mreq group;
group.imr_multiaddr = multicast_ip_address;
group.imr_interface = local_ip_address;
error =
                    setsockopt(sock,SOL_IGMP,VCRT_SO_IGMP_DROP
                    _MEMBERSHIP,
&group,sizeof(group));
```

11.15.12.19.1.5 VCRT_SO_IGMP_GET_MEMBERSHIP

IGMP get membership

Option name	VCRT_SO_IGMP_GET_MEMBERSHIP
Protocol level	SOL_IGMP
Values	
Default value	Not in a group
Change	(use with getsockopt() only; returns value in optval)
Socket type	Datagram
Comments	

11.15.12.19.1.6 OPT_RETRANSMISSION_TIMEOUT

Initial retransmission timeout

Option name	OPT_RETRANSMISSION_TIMEOUT
Protocol level	SOL_TCP
Values	>= 15 ms (See comments)
Default value	3000 (3 seconds)
Change	Before bound
Socket type	Stream
Comments	Value is a first, best guess of the round-trip time for a stream socket packet. VC/RT attempts to resend the packet if it does not receive an acknowledgment in this time. After a connection is established, VC/RT determines

	the retransmission timeout, starting from this initial value. If the initial retransmission timeout is not longer than the end-to-end acknowledgment time expected on the socket, the connect timeout will expire prematurely.

11.15.12.19.1.7 OPT_KEEPALIVE

Keep-alive timeout	
Option name	OPT_KEEPALIVE
Protocol level	SOL_TCP
Values	0 VC/RT does not probe the remote endpoint non-zero If the connection is idle, VC/RT periodically probes the remote endpoint, an action that detects whether the remote endpoint is still present
Default value	0 minutes
Change	Before bound
Socket type	Stream
Comments	The option is not a standard feature of the TCP/IP specificationand generates unnecessary periodic network traffic

11.15.12.19.1.8 OPT_MAXRTO

Maximum retransmission timeout	
Option name	OPT_MAXRTO
Protocol level	SOL_TCP
Values	non-zero Maximum value for the retransmission timers exponential backoff 0 VC/RT uses the default value, which is 2 times the maximum segment lifetime (MSL). Since the MSL is 2 minutes, the MTO is 4 minutes.
Default value	0 milliseconds
Change	Before bound
Socket type	Stream
Comments	The retransmission timer is used for multiple retransmissions of a segment.

11.15.12.19.1.9 OPT_NO_NAGLE_ALGORITHM

No Nagle algorithm

Option name	OPT_NO_NAGLE_ALGORITHM
Protocol level	SOL_TCP
Values	TRUE VC/RT does not use the Nagle algorithm to coalesce short segments FALSE To reduce network congestion, VC/RT uses the Nagle algorithm (defined in RFC 896) to coalesce short segments
Default value	FALSE
Change	Before bound
Socket type	Stream
Comments	If an application intentionally sends short segments, it can improve efficiency by setting the option to TRUE

11.15.12.19.1.10 OPT_RBSIZE

Receive-buffer size

Option name	OPT_RBSIZE
Protocol level	SOL_TCP
Values	Recommended to be a multiple of the maximum segment size, where the multiple is at least three
Default value	4380 bytes
Change	Before bound
Socket type	Stream
Comments	When the socket is bound, VC/RT allocates a receive buffer of the specified number of bytes, which controls how much received data VC/RT can buffer for the socket

11.15.12.19.1.11 VCRT_SO_LINK_RX_8021Q_PRIO

Receive Ethernet 802.1Q priority tags

Option name	VCRT_SO_LINK_RX_8021Q_PRIO
Protocol level	SOL_LINK
Values	-1 The last received frame did not have an Ethernet 802.1Q priority tag 0..7 The last received frame had an Ethernet

	802.1Q priority tag with the specified priority
Default value	
Change	(use with getsockopt() only; returns value in optval)
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received

11.15.12.19.1.12 VCRT_SO_LINK_RX_8023

Receive Ethernet 802.3 frames

Option name	VCRT_SO_LINK_RX_8023
Protocol level	SOL_LINK
Values	TRUE The last received frame was an 802.3 frame FALSE The last received frame was an Ethernet II frame
Default value	
Change	(use with getsockopt() only; returns value in optval)
Socket type	Stream (Ethernet)
Comments	Returned information is for the last frame that the socket received

11.15.12.19.1.13 OPT_RECEIVE_NOWAIT

Receive nowait

Option name	OPT_RECEIVE_NOWAIT
Protocol level	SOL_TCP
Values	TRUE recv() returns immediately, regardless of whether there is data to be received FALSE recv() waits until there is data to be received
Default value	FALSE
Change	Anytime
Socket type	Stream
Comments	

11.15.12.19.1.14 OPT_RECEIVE_PUSH

Receive push

Option name	OPT_RECEIVE_PUSH
Protocol level	SOL_TCP

Values	TRUE recv() returns immediately if it receives a push flag from the remote endpoint, even if the specified receive buffer is not full FALSE recv() ignores push flags and returns only when its buffer is full or if the receive timeout expires
Default value	TRUE
Change	Anytime
Socket type	Stream
Comments	

11.15.12.19.1.15 OPT_RECEIVE_TIMEOUT

Receive timeout	
Option name	OPT_RECEIVE_TIMEOUT
Protocol level	SOL_TCP
Values	0 VC/RT waits indefinitely for incoming data during a call to recv() non-zero VC/RT waits for this number of milliseconds for incoming data during a call to recv()
Default value	0 milliseconds
Change	Anytime
Socket type	Stream
Comments	When the timeout expires, recv() returns with whatever data that has been received

11.15.12.19.1.16 OPT_TBSIZE

Send-buffer size	
Option name	OPT_TBSIZE
Protocol level	SOL_TCP
Values	Recommended to be a multiple of the maximum segment size, where the multiple is at least three
Default value	4380 bytes
Change	Before bound
Socket type	Stream
Comments	When the socket is bound, VC/RT allocates a send buffer of the specified number of bytes, which controls how much sent data VC/RT can buffer for the socket

11.15.12.19.1.17 VCRT_SO_LINK_TX_8021Q_PRIO

Send Ethernet 802.1Q priority tags

Option name	VCRT_SO_LINK_TX_8021Q_PRIO
Protocol level	SOL_LINK
Values	-1 VC/RT does not include Ethernet 802.1Q priority tags
	0-7 VC/RT includes Ethernet 802.1Q priority tags with the specified priority
Default value	-1
Change	Anytime
Socket type	Stream (Ethernet)
Comments	

11.15.12.19.1.18 VCRT_SO_LINK_TX_8023

Send Ethernet 802.3 frames

Option name	VCRT_SO_LINK_TX_8023
Protocol level	SOL_LINK
Values	TRUE VC/RT sends 802.3 frames
	FALSE VC/RT sends Ethernet II frames
Default value	FALSE
Change	Anytime
Socket type	Stream (Ethernet)
Comments	Returns information for the last frame that the socket received

11.15.12.19.1.19 OPT_SEND_NOWAIT

Send nowait (stream socket)

Option name	OPT_SEND_NOWAIT
Protocol level	SOL_TCP
Values	TRUE Task that calls send() does not wait if data is waiting to be sent; VC/RT buffers the outgoing data, and send() returns immediately
	FALSE Task that calls send() waits if data is waiting to be sent
Default value	FALSE

Change Anytime
Socket type Stream
Comments

Send nowait (datagram socket)
Option name OPT_SEND_NOWAIT (can be overridden)
Protocol level SOL_UDP
Values TRUE
VC/RT buffers every datagram and send() or sendto() returns immediately

FALSE
Task that calls send() or sendto() blocks until the datagram has been transmitted.
Datagrams are not copied.
Default value FALSE
Change Anytime
Socket type Datagram
Comments

11.15.12.19.1.20 OPT_SEND_PUSH

Send push
Option name OPT_SEND_PUSH
Protocol level SOL_TCP
Values TRUE
If possible, VC/RT appends a send-push flag to the last packet in the segment of the data that is associated with send() and immediately sends the data. A call to send() may block until another task calls send() for that socket.

FALSE
Before it sends a packet, VC/RT waits until it has received from the host enough data is completely fill the packet
Default value TRUE
Change Anytime
Socket type Stream
Comments

11.15.12.19.1.21 OPT_SOCKET_ERROR

Socket error
Option name OPT_SOCKET_ERROR
Protocol level SOL_SOCKET

Values	
Default value	
Change	(use with getsockopt() only; returns value in optval)
Socket type	Datagram or stream
Comments	Returns the last error for the socket

11.15.12.19.1.22 OPT_SOCKET_TYPE

Socket type	
Option name	OPT_SOCKET_TYPE
Protocol level	SOL_SOCKET
Values	
Default value	
Change	(use with getsockopt() only; returns value in optval)
Socket type	Datagram or stream
Comments	Returns the type of socket (SOCK_DGRAM or SOCK_STREAM)

11.15.12.19.1.23 OPT_TIMEWAIT_TIMEOUT

Timewait timeout	
Option name	OPT_TIMEWAIT_TIMEOUT
Protocol level	SOL_TCP
Values	> 0 ms
Default value	2 times the maximum segment lifetime (which is a constant)
Change	Before bound
Socket type	Stream
Comments	Timewait timeout is the number of milliseconds that TCP waits in the timewait state

11.15.12.19.2 Example: Change send-push option to FALSE

```
uint_32 handle;
uint_32 opt_length = sizeof(uint_32);
uint_32 opt_value = FALSE;
uint_32 status;
...
status = setsockopt(handle, 0, OPT_SEND_PUSH,
&opt_value, opt_length);
if (status != VCRT_OK)
printf("\nsetsockopt() failed with error %lx", status);
status = getsockopt(handle, 0, OPT_SEND_PUSH,
&opt_value, (uint_32_ptr *)&opt_length);
if (status != VCRT_OK)
printf("\ngetsockopt() failed with error %lx", status);
```

11.15.12.19.3 Example: Change receive nowait option to TRUE

```
uint_32 handle;
uint_32 opt_length = sizeof(uint_32);
uint_32 opt_value = TRUE;
uint_32 status;
...
status = setsockopt(handle, 0, OPT_RECEIVE_NOWAIT,
&opt_value, opt_length);
if (status != VCRT_OK)
printf("\nError, setsockopt() failed with error %lx", status);
```

11.15.12.19.4 Example: Change Cecksum Bypass option to TRUE

```
uint_32 handle;
uint_32 opt_length = sizeof(uint_32);
uint_32 opt_value = TRUE;
uint_32 status;
...
status = setsockopt(handle, SOL_UDP, OPT_CHECKSUM_BYPASS,
&opt_value, opt_length);
if (status != VCRT_OK)
printf("\nError, setsockopt() failed with error %lx", status);
```

11.15.12.20 shutdown

shutdown **shut down the socket**

synopsis uint_32 shutdown(uint_32 socket, uint_16 how)

parameters

socket [IN]	Handle of the socket to shut down
how [IN]	One of (see description):
	FLAG_CLOSE_TX
	FLAG_ABORT_CONNECTION

returns VCRT_OK
Specific error code

traits Blocks, but the command is processed and returned to immediately
The application can no longer use socket

see also socket_dgram 167, socket_stream 167

description

Type of socket	Value of *how*	*shutdow* n() does the following:
Datagram	(ignored)	Shuts down *socket* immediately •Calls to *recvfro* m() return immediately •Discards queued incoming packets
Unconnected stream	(ignored)	Shuts down *socket* immediately
Connected stream	FLAG_CLOSE_ TX FLAG_ABORT_ CONNECTION	Gracefully shuts down *socke* t, ensuring that all sent data is acknowledged Calls to *sen* d() and *rec* v() return immediately If VC/RT is originating the disconnection, it maintains the internal socket context for 4 min. (twice the maximum TCP segment lifetime) after the remote endpoint closes the connection Immediately discards the internal socket context Sends a TCP reset packet to the remote endpoint Calls to *sen* d() and *rec* v() return immediately

example

```
uint_32 handle;
uint_32 status;
...
status = shutdown(handle, 0);
if (status != VCRT_OK)
printf("\nError, shutdown() failed with error code %lx",
status);
```

11.15.12.1 socket_stream

socket_stream	**create a stream socket**
synopsis	uint_32 socket_stream(void)
parameters	none
returns	Socket handle (success) VCRT_SOCKET_ERROR (failure)
traits	Blocks, although the command is serviced and responded to immediately
see also	bind 133 ()
description	The application uses the socket handle to subsequently use the socket.
example	See bind(). 133

11.15.12.2 socket_dgram

socket_dgram	**create a datagram socket**
synopsis	uint_32 socket_dgram(void)
parameters	none
returns	Socket handle (success) VCRT_SOCKET_ERROR (failure)
traits	Blocks, although the command is serviced and responded to immediately
see also	bind 133 ()
description	The application uses the socket handle to subsequently use the socket.
example	See bind 133

Part XII

12 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.

Part XIII

13 Memory Model of the VC20XX Cameras

In contrast to the ADSP2181 signal processor the TMS320C6211used in the VC20xx cameras has only one unified memory space.
There are 16Mbyte- and 64Mbyte-versions available for the SDRAM memory.

The SDRAM 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**	**64 Mbytes**
start address	0xA0000000	0xA0000000
end address	0xA0FFFFFF	0xA3FFFFFF
size (hex)	0x01000000	0x04000000

Part XIV

14 Functional Principle of the VC20XX 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 (2 channels for the VC2065), 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.

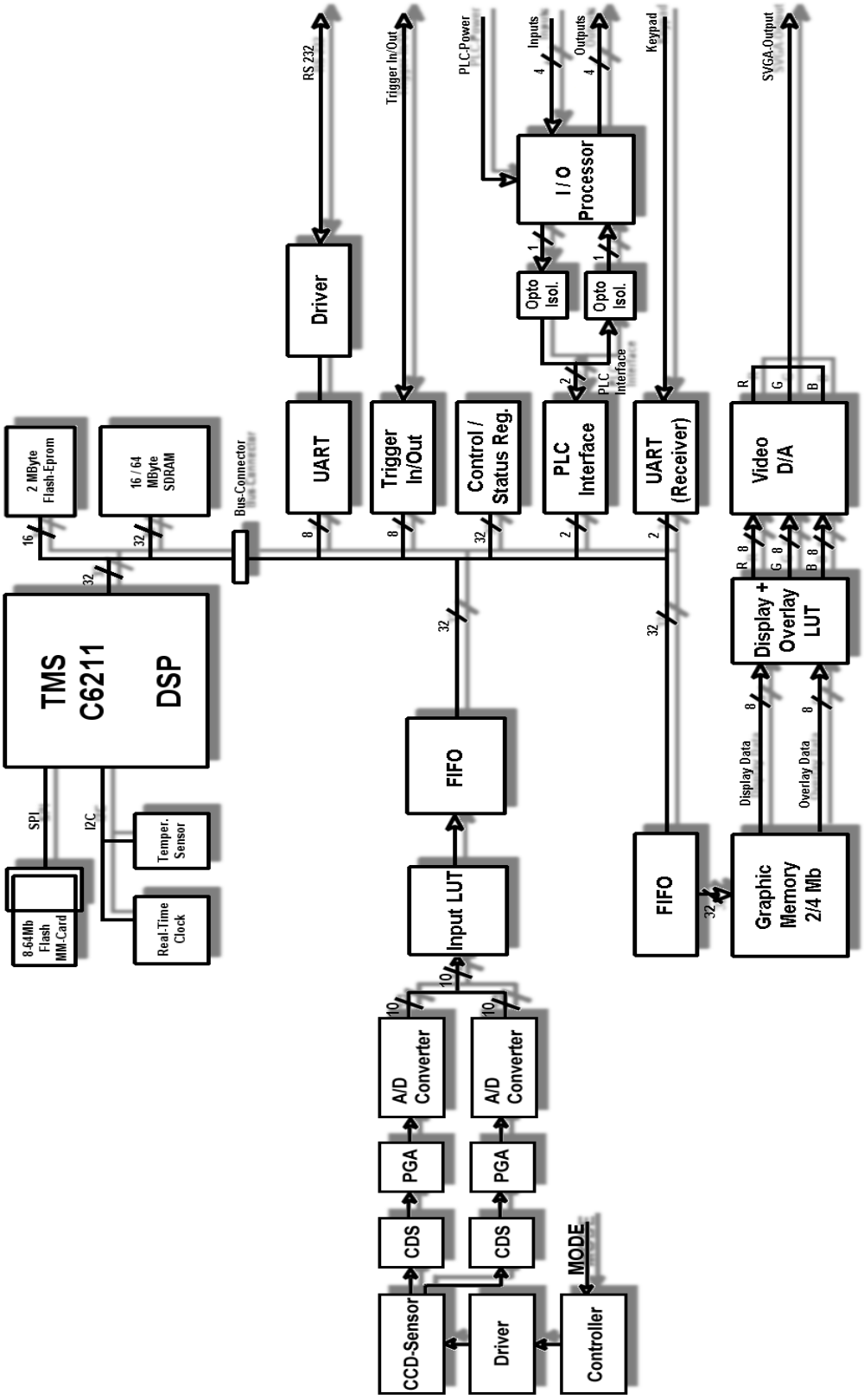
The video data may be modified using an input LUT. The image information is then stored in the DSP's main SDRAM memory using DMA.

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 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).

14.1 Block Diagram VC20xx Cameras



Part XV

15 Organization of the DRAM

The VC20xx series cameras are equipped with SDRAM (synchronous dynamic RAM) for storage of large amounts of data. The size of this SDRAM memory is 16 MBytes (or optionally 64 MBytes), organized as 4 M words à 32 bits (16 M words à 32 bits). The SDRAM is used for main memory, storing program, data and video data (images).
It is volatile, meaning the data is lost when the supply voltage is switched off.

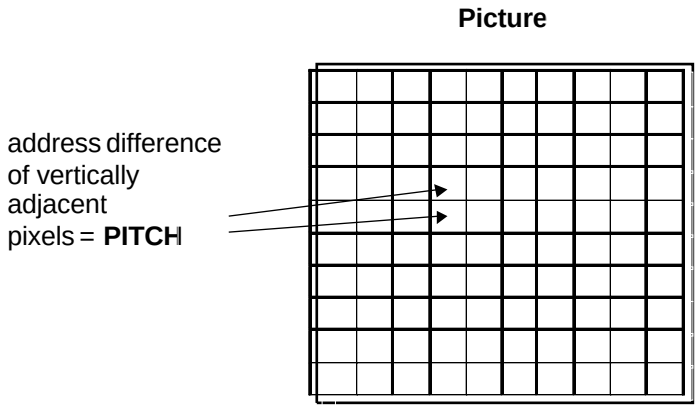
In comparison to VC series cameras using an ADSP2181 DSP, it is not necessary to use access functions for the SDRAM access. These are, however, supplied for reasons of compatibility. If there is no need for downward-compatibility, the user may easily access image SDRAM e.g. using a pointer.

Organization of the video memory:
Note , that the mapping of pixels to bytes has changed with respect to prior versions with ADSP2181 DSP. (VC20xx cameras use litlle endian byte mapping).

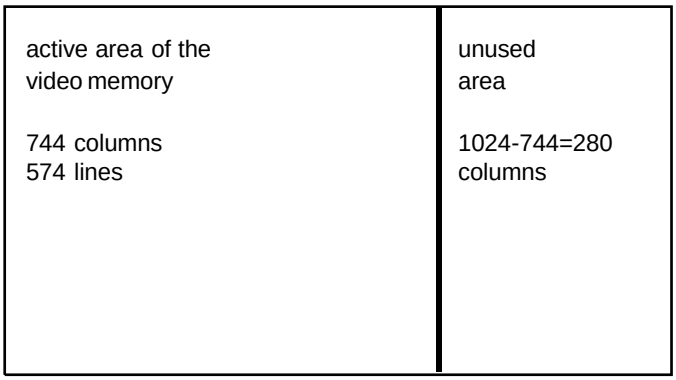
The video memory can be any part of the SDRAM. The size of this memory area depends on the frame format and the number of required frames. A start address can be specified individually for the SDRAM position of the picture taken or shown on the screen (system variables CAPT_START). or DISP_START). This 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.

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.



Part XVI

16 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, 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.

The VC2065 features powerful image graphics and overlay display features.

- 8 bit image graphics + independent 8 bit overlay
- 2 lookup tables 256x24 (RGB) for image and overlay
- 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] ¹ 0 normal overlay, display of overlay data through overlay LUT
O[7..2] = 0, O[1..0] ¹ 0 3 translucent tables, display of image data through overlay LUT

Part XVI

17 Description of the File Structure

Start address of the file system is at address 0x020000 (sector 2). User files can be stored starting at address 0x060000 (sector 6). The files are stored one after another, without gaps.

Here's the overview about the different file types :

17.1 Executable File

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	00 = executable, 01 = ASCII
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	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

The internal data structure complies to the standard .COFF format.

See ASCII File 183, Binary Data File 183, JPEG Data File 184, RLC Data File 184

17.2 ASCII File

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	01 = ASCII
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	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

See Executable File 182, Binary Data File 183, JPEG Data File 184, RLC Data File 184

17.3 Binary Data File

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	02 = Binary Data File
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	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

See Executable File 182, ASCII File 183, JPEG Data File 184, RLC Data File 184

17.4 JPEG Data File

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	03 = JPEG File
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	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

See Executable File 182, ASCII File 183, Binary Data File 183, RLC Data File 184

17.5 RLC Data File

Description	Offset	No. of bytes	Comment
Header:	0	2 bytes	ABCD
File type	2	1 byte	04 = RLC
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	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

See Executable File 182, ASCII File 183, Binary Data File 183, JPEG Data File 184

Part

18 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:

DISP_PERIOD	refresh rate for display & overlay
DISP_CNT	counter for refresh rate (counts down)
DISP_START	start address for display (must be multiple of 1024)
OVLY_START	start address for overlay (must be multiple of 1024)
DISP_ACTIVE	0: no refresh / 1: refresh (display)
OVLY_ACTIVE	0: no refresh / 1: refresh (overlay)
CAPT_START	start address for image capture (must be multiple of 1024)
HWIDTH	active horizontal pixels
VWIDTH	number of active vertical lines
VPITCH	video pitch
EXPCNT	number of exposure cycles (lines)
GAIN	video gain value
IMODE	video mode, 0=life refresh, 1=stop after current image
VSTAT	video status 0=idle 1=capture busy
INTFL	interrupt flag
CPUCLK	master cpu clock frequency
MSEC	real-time clock: millisecond
SEC	real-time clock: seconds since 1900 (long value)
EXUNIT	time unit for exposure control [usec]
DAYLIGHT	daylight savings time flag
TIMEZONE	real-time clock: timezone
LOWBAT	low battery voltage: 1=time invalid 0=time ok
TEMP	cpu board temperature
VERSION	VCRT software version
DRAMSIZE	size of main SDRAM
PLCOUT	state of the PLC outputs
PLCIN	state of the PLC inputs
POWFAIL	1: PLC power failure / 0: power ok
EXPOSING	tracking number of the image being *exposed*

THE FOLLOWING VARIABLE IS NOT AVAILABLE FOR VC/RT >= 4.0 !!!

`vline` current video line for **cameras using interlacing** , counts from 0 to 312 for the first half image and from 313 to 624 for the second one (CCIR: 625 video lines)

Please note, that most of the system variables are **highly hardware dependent** .

18.1 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 */
}
```

Part

19 C compiler

The compiler uses intelligent optimization procedures. This means the C commands are not always processed in the desired sequence. In the following, you see how the compiler handles a program that waits for a parallel process to set the semaphore a to a nonzero value:

```
int a=0;

while(a==0);
```

Compiled assembly-language program

1. Reset memory location for a to 0
2. Read memory location for a and copy to a register
3. Register = 0 ?
4. If yes, go to 3.

As can be seen, the program was compiled completely correctly but does not do what was desired. The assembly program should jump back to assembly command 2 instead of 3, to repeatedly reread the memory location for the variable. In order to get the correct result, use the statement

volatile int a=0; instead of the above.

The runtime library for the C compiler includes a series of standard functions, but **not** the function printf().

VC/RT contains a stripped-down form called print() for output of text, int and long variables but not floats. The function pstr() is used to output strings (also with the % character).

Part

20 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.

20.1 c.bat

```
cl6x -k -o3 -pl -ml3 %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 files `pgm1.cde, pgm1.obj and pgm1.int.`

The option

```
-o3
```

compiles for the best optimization possible.

```
-ml3
```

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

20.2 cc.bat

```
cl6x -k -o3 -pl -ml3 %1.c
lnk6x -u _c_int01 %1.obj -m %1.map -o %1.out cc.cmd
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 directory ..\PROCOMM.

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 and creates the S record file pgm1.msf in the directory ..\PROCOMM

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

20.3 cc.cmd

The linking process is controlled by the file cc.cmd

```
-c
-l vcrt.lib
-l vclib.lib
-l rts6201.lib
-u _c_int01
-e _c_int01
-stack 0x40000

/* for MEMORY MAP = 1 */

MEMORY
{
    PMEM:   o = 0a0200000h       l = 40000h /* intended for
initialization   */
    BMEM:   o = 0a0060000h       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 (vcrt.lib, vclib.lib, rts6201.lib)
The stack size (-stack 0x4000) and the memory map are specified
If you do not want the VCLIB to be used, or you do not own it, simply omit the string "-l vclib.lib" in cc.cmd

20.4 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 -u _c_int01 shell.obj -m shell.map -o shell.out shell.cmd
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
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 rts6201.lib or from the VCRT library.

Part

21 Description of the Example Programs

21.1 test.c

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

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

21.2 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

$
```

Part XXI

22 List of VC/RT Functions

22.1 Memory Allocation Functions

name		Type	Description
void	vcsetup(void)	M	Initialize memory management
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.
int	DRAMPagesAvail(void)	C	number of available DRAM pages
long	DRAMBytesAvail(void)	C	number of available DRAM bytes
long	DRAMWordsAvail(void)	C	number of available DRAM words
int	DRAMPgMalloc(unsigned int count)	C	allocate DRAM memory in units of a memory page
int	DRAMPageMalloc(unsigned long nbytes)	C	allocate DRAM memory in bytes, return start page of block
long	DRAMByteMalloc(unsigned long nbytes)	C	allocate DRAM memory in bytes, return start byte-address
long	DRAMWordMalloc(unsigned long nwords)	C	allocate DRAM memory in words, return start word-address
void	DRAMByteFree(long startbyte)	C	return memory block to DRAM allocation system (byte-address) void
long	DRAMWordFree(long startword)	C	return memory block to DRAM allocation system (word-address)
void	DRAMPgFree(int startpage)	C	return memory block to DRAM allocation system (page-address)
int	DRAMScreenMalloc(void)	C	allocate DRAM for full screen storage
int	DRAMOvlMalloc(void)	C	allocate DRAM for full screen overlay storage

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

22.2 Flash EPROM File Functions

Name		Type	Description
long	search(int ft, char *fname)	S	search for a file/flash EPROM
long	snext()	C	search for the next free area
long	fnaddr(long addr)	S	search for the start address of the next file/flash EPROM
int	fname(long addr, char *name)	S	get name and type of a file/flash EPROM
int	del(int ft, char *fname)	C	delete a file/flash EPROM
long	fremain()	C	remaining flash EPROM space
void	fcreat(long fp, char *name,int type)	C	create a flash EPROM file
void	fclose(long fp, long length)	C	close a flash EPROM file
int	exec(char *fname, p1,p2, ... , pn)	S	Load and execute a program from the flash EPROM
int	loadf(long addr)	S	Load program from flash EPROM (for experienced user only !)

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

22.3 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

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

22.4 DRAM Access Functions

Name	Type	Description
`int rd20(long addr)`	A	Read a word from DRAM
`void wr20(int value, long addr)`	A	Write a word to DRAM
`long rd32(long addr)`	A	read 32-bit long from DRAM
`void wr32(long value,long addr)`	A	write 32-bit long to DRAM
`int rpix(long addr)`	A	Read a byte from DRAM
`void wpix(int value, long addr)`	A	Write a byte to DRAM
`void blrdw(int count,int *buf,long addr)`	A	Read a block from DRAM, wordwise
`void blwrw(int count, int *buf,long addr)`	A	Write a block to DRAM, wordwise
`void blrdb(int count, int *buf,long addr)`	A	Read a block from DRAM, bytewise
`void blwrb(int count,int *buf,long addr)`	A	Write a block to DRAM, bytewise
`int rovl(long addr)`	A	Read the overlay bit from DRAM
`void wovl(int value, long addr)`	A	Write the overlay bit to DRAM
`void blrdo(int count, int *buf,long addr)`	A	Read a block from DRAM, bitwise (overlay)
`void blwro(int count, int *buf,long addr)`	A	Write a block to DRAM, bitwise (overlay)
`void xorpix(int value, long addr)`	A	XOR a byte in DRAM
`void xorovl(int value, long addr)`	A	XOR an overlay bit
`void blrds(int count, int *buf,long addr, int rh)`	A	read block of pixels with subsampling
`int rdrlc(int dx, int *buf, long rlc)`	A	read one line of RLC data

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

22.5 Functions Processing Pixel Lists

Name	Type	Description
`void ad_calc(int count,int *xy,long ad_list[],long start,int pitch)`	A	Address list from x/y-coordinates
`void wp_list(int count,long ad_list[], int v_list[])`	A	Write address/value list to video mem.
`void wp_set(int count,long ad_list[], int value)`	A	Write constant/address list
`void wp_xor(int count,long ad_list[], int value)`	A	XOR constant/address list
`void wo_set(int count,long ad_list[], int value)`	A	Write constant (OVL)/address list
`void wo_xor(int count,long ad_list[], int value)`	A	XOR constant (OVL)/address list
`void rp_list(int count,long ad_list[], int v_list[])`	A	Read video memory/address list
`void wo_list(int count,long ad_list[], int v_list[])`	A	Write address/value list to overlay
`void ro_list(int count,long ad_list[], int v_list[])`	A	Read overlay/address list

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

22.6 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
`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)`	C	select "lossy" external trigger mode
`void SET_trig_sticky(void)`	C	select "sticky" external trigger mode

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

22.7 RS232 Basic Functions

Name	Type	Description
`void rs232snd(char c)`	S	Output a character/serial interface
`char rs232rcv()`	S	Read a character/serial interface
`int sbfull()`	C	send buffer full/serial interface
`int sbready()`	S	send buffer ready/serial interface
`int rbempty()`	C	receive buffer empty/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

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

22.8 Basic Flash EPROM Access Functions

Name	Type	Description
`int getf8(long addr)`	M	low-level function for reading a byte/flash EPROM
`int getf16(long addr)`	M	low-level function for reading a 16-bit word/flash EPROM
`long getf32(long addr)`	M	low-level function for reading a 32-bit word/flash EPROM
`int flpgm(long addr, int value)`	M	low-level function for writing a byte/flash EPROM **(reversed args !)**
`int flpgm8(int value, long addr)`	S	low-level function for writing a byte/flash EPROM
`int flpgm16(int val, long addr)`	C	low-level function for writing a word (16 bits) to flash EPROM
`int flpgm32(long val, long addr)`	C	low-level function for writing a long-word (32 bits) to flash EPROM
`int erase(int sector)`	S	low-level function for erasing sectors/flash EPROM
`void bdma(unsigned int count,int mode, int laddr, long addr)`	S	copy flash EPROM to DMEM,via BDMA

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

22.9 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)
`int getstptr()`	A	Read stack pointer
`int getdp()`	A	Read data pointer
`int getbss()`	A	read start of bss

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

22.10 Lookup Table 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_lut_comp(int r, int g, int b)`	C	LUT compatibility mode
`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

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

22.11 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
`void RTC_set_time(void)`	C	Program Real Time Clock Chip

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

Index

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

- X -