Pixel/Strips DSP Card Software

B. Schmid
University of California, Irvine

June 22, 1995

Abstract

This document describes the DSP software, the communication protocol between the VME-CPU and the DSP card, the DSP code for the UC Santa Cruz strips Low Level card, and the DSP code for the LBL pixels Low Level card.

Introduction

The DSP cards reside in the VME crate. They are interfaced to the VMEbus. Communication between the VME-CPU and the DSP's is established by the exchange of messages. The software is coded in C-language. Note: the code described in this document is running on the DSP.

Updates June 22, 1995

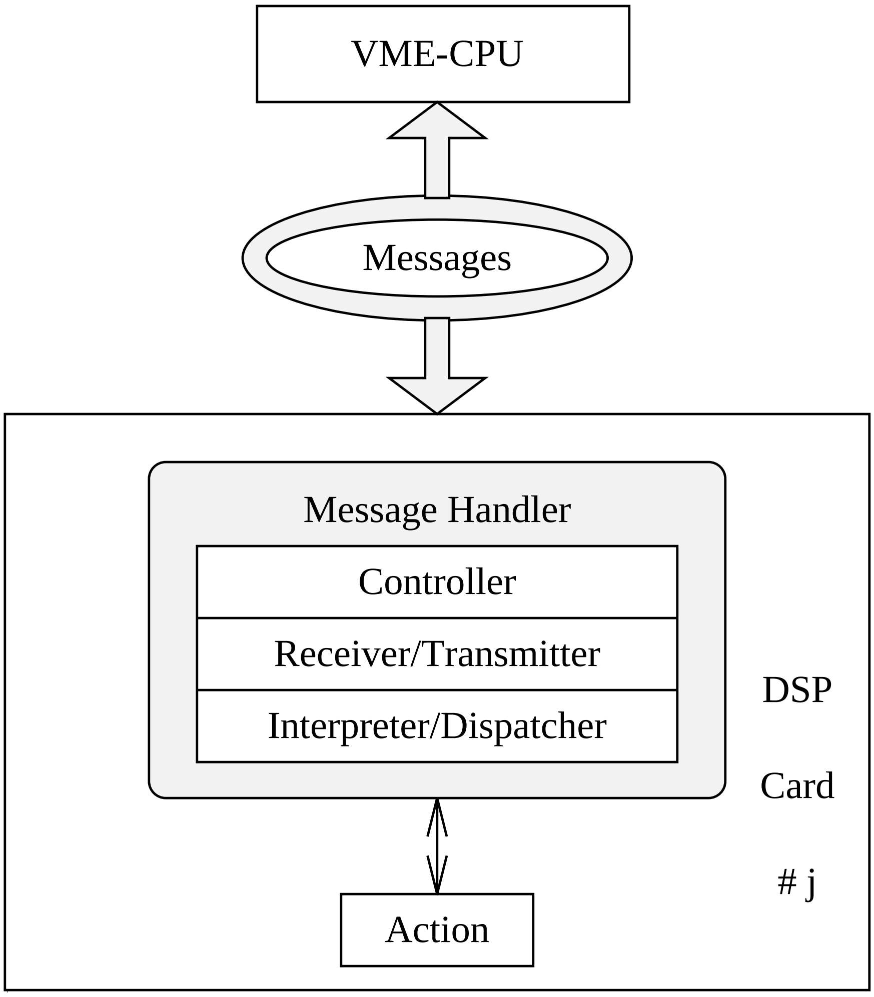
- system, strips, pixels: added section default settings
- pixels: added/deleted messages, added error codes updated description of messages, functions
- strips: added message, updated description of messages, functions
- user: updated descriptions/requirements
- system: updated descriptions

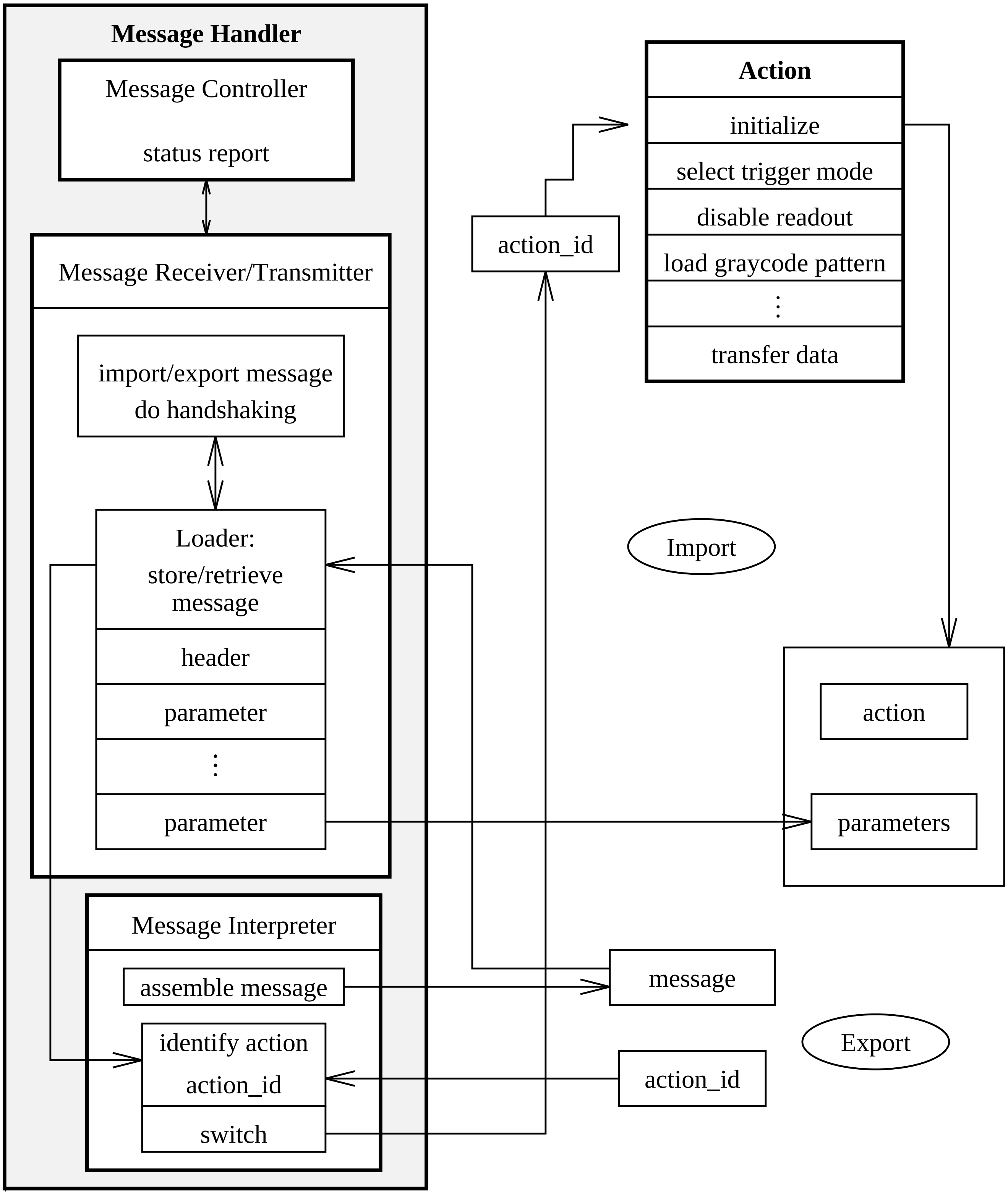
Contents

1 Concept

1.1 Overview

Communication between the VME-CPU and the DSP's is established by the exchange of messages.




Message Handler
Message Controller
status report
Message Receiver/Transmitter
import/export message
do handshaking
Loader:
store/retrieve
message
header
parameter
⋮
parameter
Message Interpreter
assemble message
identify action
action_id
switch
action_id
Action
initialize
select trigger mode
disable readout
load graycode pattern
⋮
transfer data
Import
action
parameters
message
Export
action_id

1.2 Messages

Messages are transmitted to the dual port memory of the VME-interface (IO-memory) which is part of the DSP card. The IO-memory is mapped and accessible from both the VME-CPU and the DSP. Handshaking between the VME-CPU and the DSP is done by means of setting/checking flags which are stored in the IO-memory at addresses beginning at IO_FLAGS_BASE. The message format is given in the table below. The header consists of a command identifier word *ciw*, two DSP identifiers *dsp* and *dsp_dipsw*, an identifier *channel* for the channel serviced by the DSP, a host action_id identifier *host_ciw* indicating which host message caused the DSP message, and the message length *len*. Additional header words may be added. The command identifier word (ciw) specifies the message type and the requested function (given by the *action_id*) to be performed. The ciw consists of a x-bit flag and the (32-x)-bit action_id. For the time being, the flag is 0 and *ciw = action_id*. The message length specifies the length (number of words) of the whole message, including the header. Parameters follow the header. All words are 32-bit words.

Message Format

header
parameter 1
...
parameter N

Header:

command identifier word
message length
DSP #
DSP Dip Switch Setting
channel
host_ciw
spares[5]

command identifier word: flag + action_id; flag = 0

Messages are coded as structures. Each structure has the same header variables (*ciw, len, dsp, dsp_dipsw, channel*, and *spare[5]*). The header variables are followed by the message parameters. These structures can be accessed by defining a pointer *mpt* of type *struct struct_name*
*mpt = ((struct struct_name *) p_mess)*. The pointer *p_mess* is globally defined.

A flag VERBOSE_FLAG, settable by the VME-CPU, enables/disables the DSP to respond to messages from the VME-CPU. If VERBOSE_FLAG = 1, the DSP sends the response message *response* before or after execution of the action requested by the VME message (unless the action requested results in sending a message to the VME). The response message has a response code as its parameter (see page 25). The verbose flag is initialized to one by the DSP.

1.3 Boot Process

The DSP's boot loader loads the boot code from the EEPROM into the appropriate section of the SRAM. The boot code is then executed: the DSP reads blocks of code, passed by the VME-CPU, from the Dual Port RAM and loads them into the appropriate sections of the internal RAM and the SRAM. It then starts execution of the code loaded which includes turning on the green LED and setting the flags CODEVER_SYS and CODEVER_SCOM_US to the the version number of the code. The DSP then calls the function *initialize* before it goes into the main loop (see pages 8ff). If, during initialization, the DSP detects that the High Level Card (VME Interface) is not connected, it turns on the red LED. The DSP sends the error message *fatal_error* with the error code NO_LL if the Low Level Card is not connected.

1.4 Major DSP States, Start of Burst and End of Burst Signals

There are two major DSP states: handling messages while waiting for a start of burst signal and reading trigger and detector data until an end of burst signal occurs. The readout is aborted by a fatal error in which case an error message is sent to the VME. The transfer of data acquired prior to the fatal error can be requested. To resume data taking, a start of burst signal has to be applied again. Previously acquired data will be lost. A flag DSP_STATE indicates whether the DSP is idle (waiting for messages and a start of burst signal), is executing an action, or whether it is in the 'burst-loop' reading data and waiting for the end of burst signal. These states are visualized by the LED's on the DSP card. A steady green light indicates the idle state, the yellow light is turned on during execution of an action induced by a message, the green light is flashing during the burst loop, and the red light indicates a fatal error. The major DSP States are shown on pages 11ff.

The start of burst and end of burst signals can be sent to the DSP card through its direct input port. In addition, software controlled data taking is possible by setting and resetting flags which correspond to the the start of burst and end of burst signals. Software control of data taking requires the following procedure: the start of burst flag SOB_SIG_FLAG has to be cleared (set to 0) before the end of burst flag EOB_SIG_FLAG can be set (set to 1). The end of burst flag has to be cleared before the start of burst flag can be set. A calibration run does not require any external (hardware or VME software) start of burst and end of burst signals. The burst flags are initialized to zero by the DSP. The various flags are defined in section 2.3.

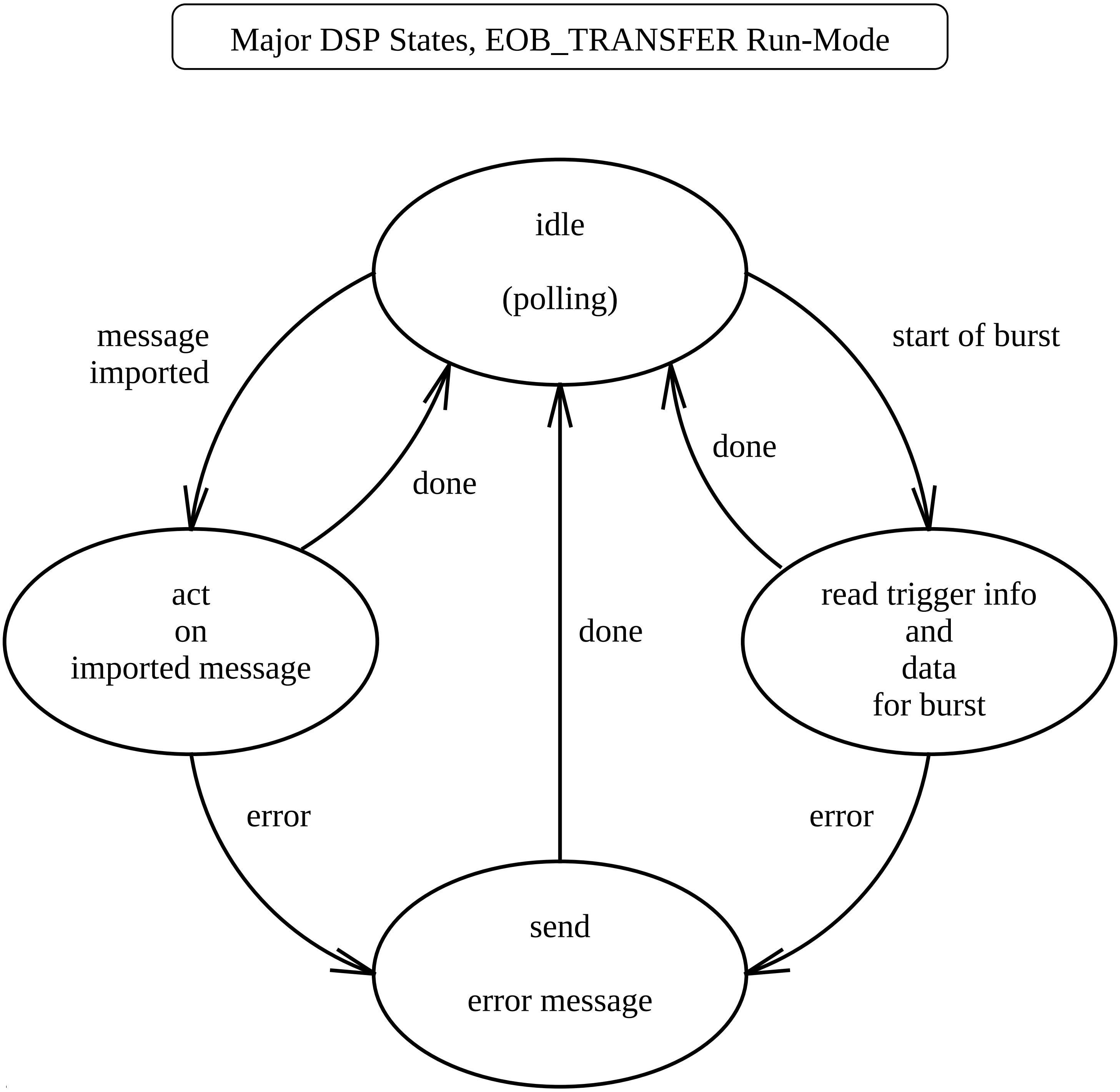
1.5 Run Modes and Data Transfer

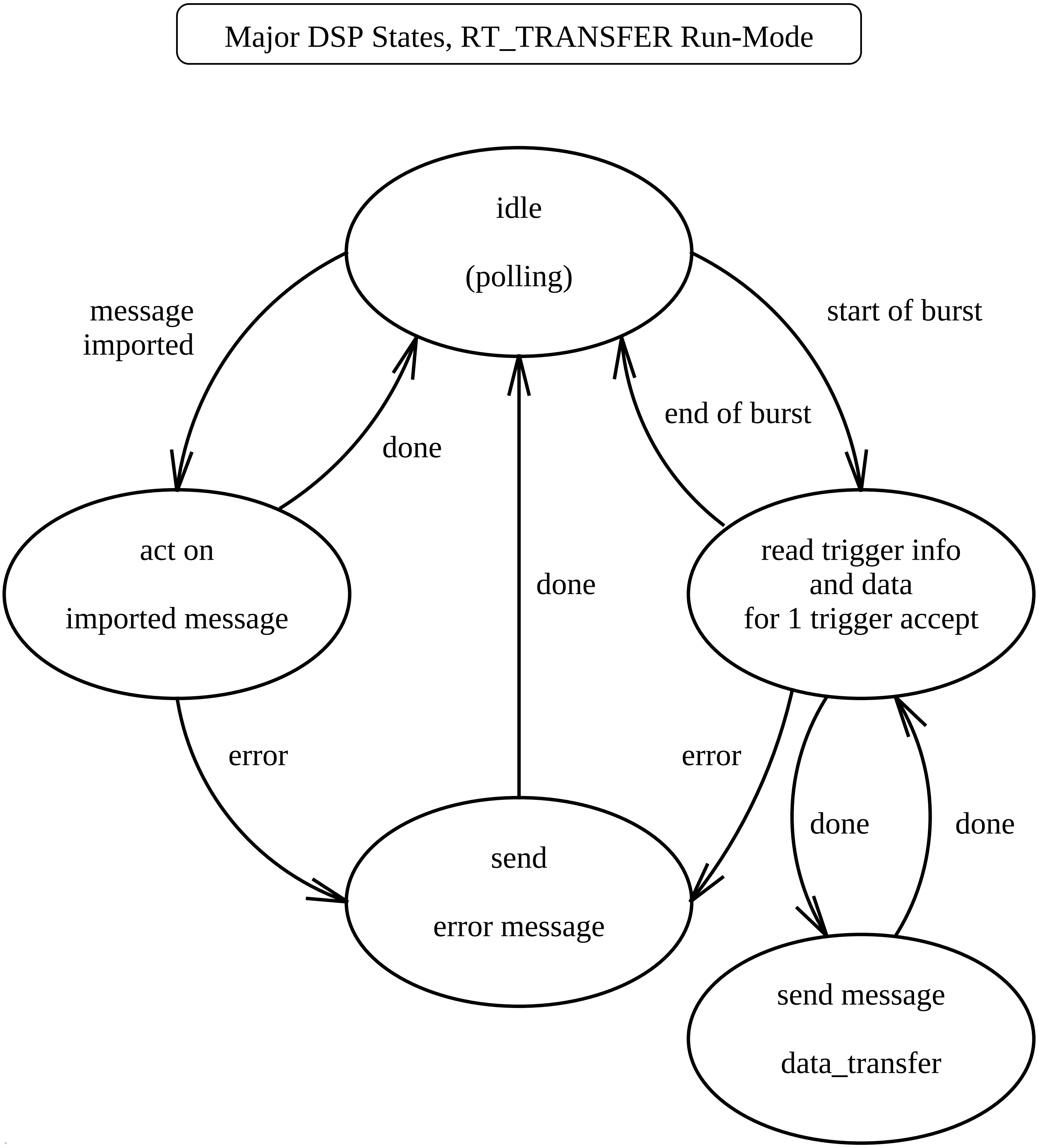
There are two run-modes possible: data transfer at the end of a burst (EOB_TRANSFER mode) or transfer of data event by event during the burst (RT_TRANSFER mode). The run-mode is set to the default value during initialization. The mode can be changed by the message *select_run_mode*.

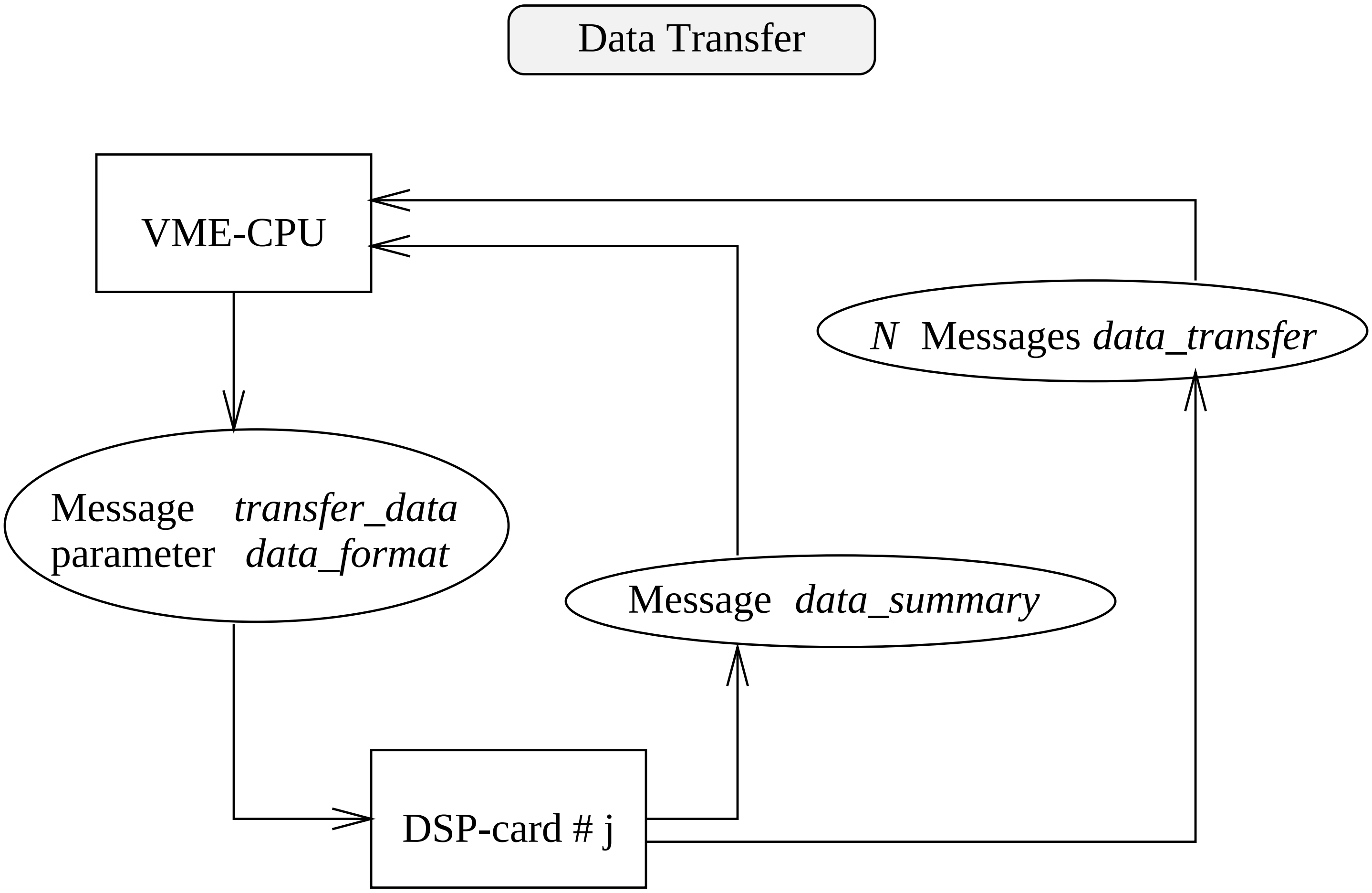
In EOB_TRANSFER mode it is the VME-CPU's responsibility to request the transfer of the data from the DSP card to the VME-CPU. Data transmission is initiated by sending the message *transfer_data* to the DSP card. Three data formats are supported: (1) ordered data, *i.e.* blocks of trigger information and the corresponding detector data are transmitted in messages *data_transfer* (one message per trigger), (2) raw detector data, *i.e.* the complete block of detector data from one link (all links if the message header field *channel* is less than zero) as stored in memory is transmitted in one single message *data_transfer* (NUMLINKS messages if the message header field *channel* is less than zero), and (3) raw trigger data, *i.e.* the complete block of trigger data as stored in memory is transmitted in one single message *data_transfer*. The user can define additional data formats. The data format is selected by the parameter *data_format* which is submitted by the message *transfer_data*. If the data format is negative, the DSP sends the message *data_summary* prior to transmitting the data. The data summary provides a data consistency check. It transmitts the number of triggers received (which is set to 0x800000 in the case of missing trigger information (for DSP cards without trigger control logic)) and the number of data blocks read out. A flag is provided to transmit other information of data validation. Upon receipt of the request *transfer_data*, the DSP sends either the message *data_summary* followed by messages *data_transfer* or transmits the data in messages *data_transfer*.

In RT_TRANSFER mode, the VME-CPU has to set the flag TRIGGER_FLAG upon receipt of a trigger. In this mode, the DSP automatically sends the message *data_transfer* in format (1) event by event. After transmission, the data is overwritten with data from the next event. If no detector data has been received within a time period specified by the parameter *rt_readout* of the message *select_time_param*, the DSP sends the message *error_report* with the error code NO_DET_DATA and resets the TRIGGER_FLAG. This flag is also reset by the DSP before it sends the message *data_transfer*. The DSP will not set any *BUSY* signal. It is the VME-CPU's responsibilty to set and reset any necessary *BUSY* signals.

Data transfer in a calibration run is identical to the EOB_TRANSFER mode. Calibration is initiated by the message *calibrate*. The number of triggers to acquire is set by the message parameter *acqui_events*. The DSP will control the start of burst and end of burst signals. The DSP generates the triggers and calibration pulses (pixels) or executes a control block (strips). Readout of the detector is enabled by setting the parameter *take_data* to one. At the end of the calibration cycle the DSP sets the flag DSP_DATA_READY_FLAG to one and waits for messages from the VME-CPU. The calibration run can be aborted by setting the flag ABORT_FLAG to one. The DSP will terminate the calibration run as described above.


Major DSP States, EOB_TRANSFER Run-Mode
idle
(polling)
message
imported
start of burst
done
done
act
on
imported message
done
read trigger info
and
data
for burst
error
error
send
error message


Major DSP States, RT_TRANSFER Run-Mode
idle
(polling)
message imported
start of burst
end of burst
done
act on
imported message
done
read trigger info
and data
for 1 trigger accept
error
error
done
done
send
error message
send message
data_transfer



data_summary:

HEADER
flag
# of trigger blocks
# of data blocks

data_transfer:

HEADER
data_format
data

The number of trigger blocks is set to 0x80000000 (bit MSb 31 set) if there is no trigger information. The meaning of the flag is defined by the user.

Data

The trigger information for each trigger is packed into two 32-bit words:

Trigger Record 1

31					16	8	0
1	xxxxxxx	xxxxx	type	type	type	xxxxxxxx	timestamp

Trigger Record 2

31 … 0
beam crossing number

where *type* specifies the trigger source (four different sources are possible).

data_format = 1: 1 message per trigger

Message # i
Trigger Record i 1
Trigger Record i 2
Detector Data for Trigger # i

data_format = 2: 1 message per link; if *channel* < 0: NUMLINKS messages

Block of Detector Data from Link #j

data_format = 3: 1 message total

Block of Trigger Data

data_format > 3: defined by the user.

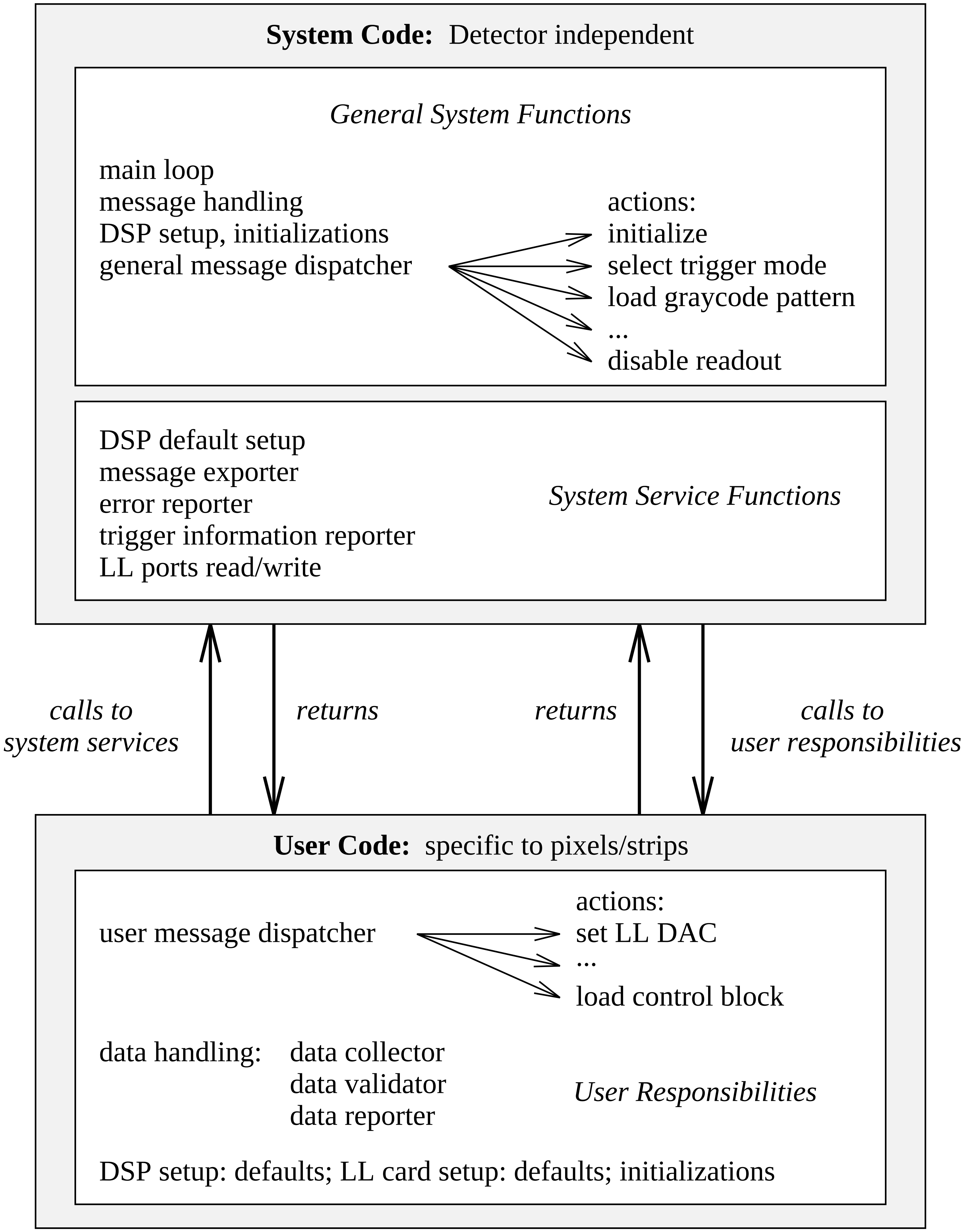
1.6 DSP Code

The DSP code includes system code and user code. System code is independent of the detector, whereas user code is specific to the detector interfaced to the DSP card through the low level card. System code includes:

- boot code
- main loop
- VME $\leftrightarrow$ DSP communication software
- functions for DSP card setup and initializations
- framework for data collection
- framework for data reporting
- interface to user code
- service functions

User code has to provide functions for:

- low level card setup and initializations
- message dispatching
- data handling


System Code: Detector independent
General System Functions
main loop
message handling
DSP setup, initializations
general message dispatcher
actions:
initialize
select trigger mode
load graycode pattern
...
disable readout
DSP default setup
message exporter
error reporter
trigger information reporter
LL ports read/write
System Service Functions
calls to system services
returns
returns
calls to user responsibilities
User Code: specific to pixels/strips
user message dispatcher
actions:
set LL DAC
...
load control block
data handling:
data collector
data validator
data reporter
User Responsibilities
DSP setup: defaults; LL card setup: defaults; initializations

2 System Code

System code is grouped into the following sections:

- General Messages
- System functions invoked by general messages
- System service functions: setup, message handling, action dispatching, trigger handling, and status reports.

Some system functions invoke functions which have to be coded by the user. A compilation of these user functions is given in the section *General User Code*.

System code is defined in the following files:

- *sys.h*: structure definitions of messages, definitions of the corresponding action_id's, definitions of flags, error and response codes, definitions of base-addresses, ports, trigger control, and definitions of function prototypes.
- *sys_glob.h*: definitions of global pointers and variables.
- *sys_ext.h*: declarations of external variables.
- *sys_main.c*: main program.
- *sys.c*: system functions.
- *sys_com.h*: header file for communication code.
- *sys_com.c*: VME ↔ DSP communication code.
- *sys_boot.c*: boot code.

Utility functions are provided in the files *sys_util.h* and *sys_util.c*.

2.1 List of System Messages

List of General Messages: VME → DSP			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
calibrate	m_calib	3	initiates calibration run
		acqui_events	number of calibration events
		take_data	enables/disables detector readout
		user_par	user parameter strips: control block [0-3] to execute
disable_readout	m_disable_read	none	disables detector readout
enable_readout	m_enable_read	none	enables detector readout
echo	m_echo	*	requests message echo_reply
		data[*]	parameters *: MAX_MESS_ARRAY
get_dsp_status	none	none	requests DSP setup parameters, DSP message dsp_status
initialize	m_init	none	initializations, default settings
load_graycode_pattern	m_load_gray	256	loads graycode pattern
		graycode[256]	data
load_data	m_load_data	*	loads data into SRAM for link *channel* *channel* < 0: all links
		data_format	data format (2 or 3)
		data_word[*]	data, *: MAX_MESS_ARRAY
message_test	m_message_test	3	requests message test_message
		tmess_size	number of parameters
		startval	starting value for parameters
		delta	increment

List of General Messages: VME → DSP			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
select_run_mode	m_sel_run	1	selects run-mode/data transfer
		runmode	0: data transferred after end of burst 1: data transferred during burst
select_trigger_mode	m_sel_trig	4	selects trigger mode
		trignum[4]	# of accepts/trigger for source [0-3]
select_align_delay	m_align_delay	2	selects delays for alignment FIFO's
		delay1	trigger alignment FIFO 1 delay [clocks]
		delay2	trigger alignment FIFO 2 delay [clocks]
set_dac	m_set_dac	2	sets DSP card DAC
		dac	destinaton DAC (0-3)
		voltage	voltage (0-4095)
select_time_param	m_sel_time	3	selects timing parameters
		tr_time	time interval to read trigger counter and FIFO [μs]
		rt_rtout	timeout for readout routine (RT_TRANSFER mode) [μs]
		t_addoff	offset for 13 bit graycode address (trigger FIFO word)
transfer_data	m_transfer_data	1	sends data summary to VME and transfers data to VME
		data_format	data format
validate_data	none	none	requests DSP message data_summary

DSP responses:

- calibrate ⇒ ACCEPTED (if flag VERBOSE_FLAG = 1)
- echo ⇒ DSP message *echo_reply*
- get_dsp_status ⇒ DSP message *dsp_status*
- message_test ⇒ DSP message *test_message*
- transfer_data ⇒ DSP messages *data_summary, data_transfer*
- validate_data ⇒ DSP message *data_summary*
- all other messages ⇒ DONE (if flag VERBOSE_FLAG = 1)

List of General Messages: DSP → VME			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
data_transfer	none	1 + ?	transfers data to VME
		data_format	data format
		data[?]	
data_summary	m_data_summary	3	data summary
		flag	meaning of flag is user defined
		numtrig	# of trigger records = 0x80000000: no trigger information
		numdat	# of data records
dsp_status	m_dsp_status	14	response to get_dsp_status
		dac_setting[4]	DSP card DAC settings
		al_delay_setting[2]	TCC trigger alignment FIFO delays
		runmode	RT_TRANSFER of EOB_TRANSFER
		ro_enabled	detector readout enabled/disabled
		tcc_connected	trigger card TCC connected
		tr_time	time interval for reading trigger info
		rt_rtout	timeout for readout routine
		t_addoff	offset for 13 bit graycode address
		trig_timer_val	current value of trigger counter
		clock_timer_val	current value of timer
error_report	m_error_report	1	general error message
		error_code	error code
fatal_error	m_fatal_error	1	error message
		error_code	indicates source of error
response	m_response	1	DSP response to VME-CPU message
		response_code	response code ('done', 'accepted')
echo_reply	none	variable	DSP response to echo message
test_message	none	variable	response to message_test

2.2 Definitions of System action_id

Numbers 1-99 are reserved for system action_id's, numbers 100-199 for pixel-code action_id's, and numbers 200-299 for strips-code action_id's. DSP error messages have action_id's starting at 80.

Definitions of action_id	
General Messages/Actions initiated by VME-CPU	
Definiton	***Value***
CALIBRATE	16
DISABLE_READOUT	3
ENABLE_READOUT	4
ECHO	19
GET_DSP_STATUS	21
INITIALIZE	1
LOAD_DATA	15
LOAD_GRAYCODE_PATTERN	12
MESSAGE_TEST	20
SELECT_ALIGN_DELAY	11
SELECT_RUN_MODE	17
SELECT_TIME_PARAM	18
SELECT_TRIGGER_MODE	5
SET_DAC	9
TRANSFER_DATA	6
VALIDATE_DATA	22

Definitions of action_id	
General Messages sent from DSP to VME	
DATA_SUMMARY	51
DATA_TRANSFER	50
DSP_STATUS	57
ECHO_REPLY	55
RESPONSE	53
TEST_MESSAGE	56
Error messages sent from DSP to VME	
ERROR_REPORT	81
FATAL_ERROR	80

2.3 Definitions of System Flags

The following flags are accessible by the VME-CPU and the DSP by calling the functions *get_flag(int flagnum)* and *set_flag(int flagnum, int flagval)*. The flag value true (false) corresponds to a value $\neq 0$ (0). Flag numbers 0-20 are reserverd for system code, numbers 21-40 for pixel user code, and numbers 41-64 for strips user code.

Definitions of Flags		
Definiton	***Flag #***	***Description***
ABORT_FLAG	5	allows halting of diagnostics runs
BOOT_CODE_VERSION	7	version number of boot code byte 3 (MSB): sys.h, byte 2: sys_glob.h byte 1: empty, byte 0 (LSB): sys_boot.c
CODEVER_SYS	9	version number of system code byte 3 (MSB): sys.h, byte 2: sys_glob.h byte 1 sys.c, byte 0: sys_main.c
CODEVER_SCOM_US	10	version number of system communication and user code byte 3: sys_com.h, byte 2: sys_com.c byte 1: user.h, byte 0: user.c
DSP_DATA_READY_FLAG	3	set at the end of a calibration run
DSP_STATE	4	DSP status flag: RESET_COMPLETE (4), DOWNLOAD_COMPLETE (5), IDLE (1), EXECUTING (2), BURST_LOOP(3) 1: DSP is waiting for messages and start of burst 2: DSP is executing action initiated by message 3: DSP is reading data and waiting for end of burst 4: reset done, boot code is running 5: code downloaded and running values 20 - 60: states reserved for strips code values 61 - 100: states reserved for pixel code
EOB_SIG_FLAG	1	software controlled end of burst signal
SOB_SIG_FLAG	0	software controlled start of burst signal
HOST_STATE	8	host status flag HS_RESETTING_DSP (0xfadedbad): host is resetting DSP HS_READY (0): host ready
TRIGGER_FLAG	2	set by VME-CPU when trigger occurred (in RT_TRANSFER run mode)
VERBOSE_FLAG	6	enables/disables sending of DSP response message initialized to 1 (true) by DSP

DSP/VME Access of Flags		
Flag	***DSP Access #***	***VME Access***
ABORT_FLAG	R	R/W
BOOT_CODE_VERSION	R/W	R
CODEVER_SYS	R/W	R
CODEVER_SCOM_US	R/W	R
DSP_DATA_READY_FLAG	R/W	R
DSP_STATE	R/W	R
EOB_SIG_FLAG	R/W (W calibration run)	R/W
SOB_SIG_FLAG	R/W (W calibration run)	R/W
HOST_STATE	R	R/W
TRIGGER_FLAG	R/W (RT_TRANSFER mode)	R/W
VERBOSE_FLAG	R/W (W at initialization)	R/W

2.4 Definitions of System Error and Response Codes

Error codes are passed by the DSP messages *fatal_error* and *error_report*, response codes by the DSP message *response*. Numbers 1-99 are reserved for system codes, numbers 100-199 for pixel-code, and numbers 200-299 for strips-code.

Definitions of Error and Response Codes		
Definiton	***Value***	***Error/Response***
Fatal Errors		
COMMAND_UNKNOWN	9	message command identifier unknown
FORMAT_UNKNOWN	10	requested data format unknown
GRAYCODE_TOO_LONG	7	graycode block too long
INVALID_RUNMODE	5	invalid run mode
INVALID_AL_DELAY	6	invalid trigger alignment delay value
INVALID_LINK	8	invalid link number
MEM_OVERFLOW	4	memory overflow
MESSAGE_TOO_LONG	11	message sent too long
NO_LL	1	low level card not connected
RO_ERROR	3	detector readout error
Other Errors		
INVALID_DAC_VALUE	50	DAC value out of range
INVALID_DESTINATION_DAC	51	invalid destination DAC
NO_DATA	55	no data at all
NO_DET_DATA	53	no detector data
NO_TRIG_DATA	54	no trigger data
TRIG_ERROR	52	error reading trigger information
Response Codes		
ACCEPTED	1	message received, action accepted
DONE	0	message received, action executed
REFUSED	2	message received, action refused

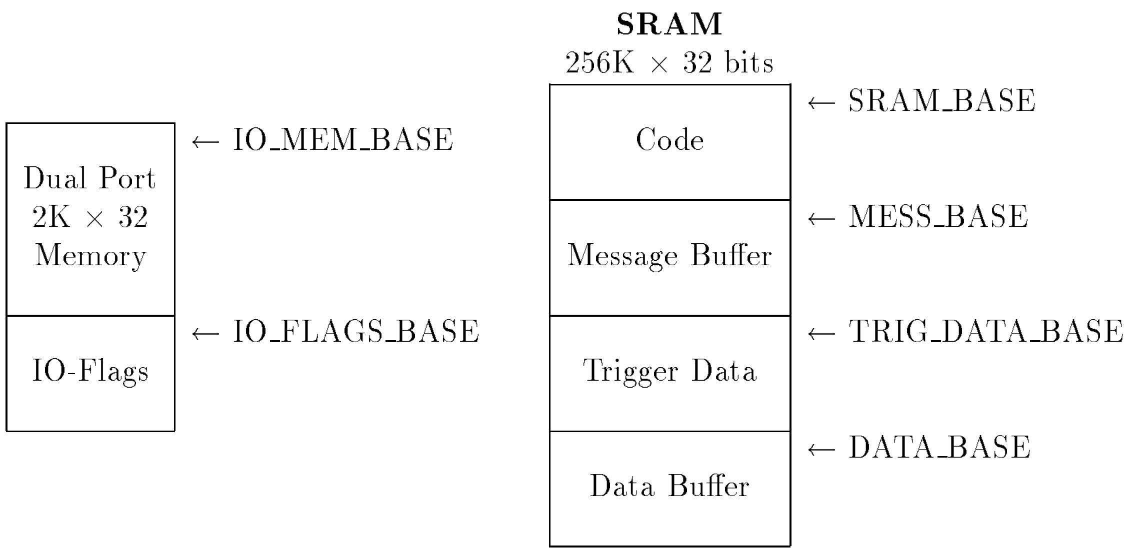
2.5 System Default Settings

System default values are set after the DSP code has been downloaded and by the message *initialize*.

System Default Settings		
Message	***Parameter***	***Default Value***
select_run_mode	runmode	EOB_TRANSFER
	VERBOSE_FLAG	1
enable/disable_readout	readout_enabled	1
select_trigger_mode	trignum[4]	4∗1
select_align_delay	delay1 [clocks]	80
	delay2 [clocks]	80
set_dac	voltage	4∗0
select_time_param	tr_time [μs]	80
	rt_rtout [μs]	200
	t_addoff	0

2.6 Definitions of Base Addresses, External Pointers and Variables

2.6.1 Base Addresses



CODE_BLOCKSIZE:	size of code block:	0x2000 + 0x800 for boot code
MESS_BLOCKSIZE:	size of message buffer:	0x2000
TRIG_BLOCKSIZE:	size of trigger data buffer:	0x2000
DATA_BLOCKSIZE:	size of data buffer:	0x39800

2.6.2 Globally Defined Pointers

m_generic **p_mess:*	points to *structure m_generic ((struct m_generic *)* MESS_BASE)
int **p_data:*	points to DATA_BASE
int **p_trig:*	points to TRIG_DATA_BASE
int **p_vmeio:*	points to IO_MEM_BASE

For more details and the addresses of memory mapped input and output ports refer to section B.1 (sys.h, sys_glob.h).

2.6.3 Globally Defined Variables

struct ports port[SYS_MAXPORTS]	system input and output ports
int data_words	# of data words written into data buffer
int data_blocks	# of data blocks written into data buffer
int trig_words	# of trigger words written into trigger data buffer
int trig_blocks	# of trigger blocks written into trigger data buffer
int run_mode	mode of data acquisition/data transfer
int acqui_events	# of calibration events to generate
int tcc_exist	flag indicating whether trigger control card is attached
int readout_enabled	flag indicating whether readout is enabled
int trigread_time	time after which trigger FIFO has to be read out
int rt_read_timeout	timeout for readout loop in realtime transfer mode
int t_add_offset	offset to be added to 13 bit graycode address due to delay (address part of trigger FIFO word)
int header_dsp	DSP number, addressed in message from VME-CPU
int header_dsp_dipsw	DSP's dip switch setting
int header_channel	channel addressed in message from VME-CPU
int al_delay[2]	trigger alignment FIFO's delays [clocks]
int sys_dac_val[4]	DSP card DAC[0-3] values
int calib_take_data	data taking (true/false) in calibration run
int host_action	action_id of VME message

For more details see section B.1 (sys.h, sys_glob.h).

2.7 System Ports

The system ports are defined as an array of structures:

```
struct ports {
   volatile int *address;
   volatile int current_values;
};
struct ports port[SYS_MAXPORTS];
```

The system ports can be accessed by the functions

- int writebit_port(int portnum, int bitnum, int bitval)
- int write_port(int portnum, int word)
- int readbit_port(int portnum, int bitnum)
- int read_port(int portnum, int portmask)

The port numbers *portnum*, port masks *portmask*, and the port addresses are defined in *sys.h*. The array *port*[SYS_MAXPORTS] is initialized in *sys_glob.h*. For more details see

Port	***Port number***	***Port Address***	***Input Port Mask***
DAC, LED Port	OUT_PORT0	A_OUT_PORT0	
Direct Inputs, Interrupt Inputs	IN_PORT0	A_IN_PORT0	M_IN_PORT0
Cards Present Bits, DIP Switch Bits	IN_PORT1	A_IN_PORT1	M_IN_PORT1
Low Level Card	LL_PORT	A_LL_PORT	
High Level Card	HL_PORT	A_HL_PORT	
Trigger Control	TRIG_CONT	A_TRIG_CONT	
Trigger FIFO Status Flags	TRIG_STAT	A_TRIG_STAT	M_TRIG_STAT
Trigger FIFO	TRIG_DATA	A_TRIG_DATA	M_TRIG_DATA
Trigger Graycode Counter	TRIG_GRAY	A_TRIG_GRAY	
Trigger Pulse Generation	TRIG_MPUL	A_TRIG_MPUL	

section B.1 (sys.h, sys_glob.h).

2.7.1 Bit Definitions for Control of DAC's and LED's

7							0
LDAC	DAC_CS	DAC_SCK	DAC_SDI	x	LED_G	LED_Y	LED_R

The function call *writebit_port*(OUT_PORT0, LDAC, 0) will set bit 7 of the DAC/LED port to 0.

2.7.2 Bit Definitions for Trigger Control and Status Ports

Trigger Control Port:

7							0
AL_PAUSE	AL_RES	GR_MODE	GR_WE	GR_LOAD	GR_HOLD	TF_NEXT	TF_RES
15							8
TRG_I3	ctrl_spare					AL_INJ_SEL	AL_INJ

Trigger FIFO Status Flags Port:

15			0
stat_spare	TF_F2	TF_F1	TF_EF

The function call *read_port*(TRIG_STAT, M_TRIG_STAT) will return the value of the Trigger Status Flags Port.

2.8 List of System Service Functions

System service functions are defined in *sys.c*, *sys.h*, *sys_util.h*, and *sys_util.c* (LL ports read/write). Unless specified otherwise, a function is defined as *int function(void)*.

List of System Service Functions
Setup and Initialization:
set_sys_defaults(int *, int, int, int, int, int)
Message Handling:
export_message(int *pointer)
assemble_header(int, int, int *)
send_response(int)
report_error(int action_id,int error_code)
Data Collecting:
exe_trig_calib
report_error(int action_id,int error_code)
Data Reporting:
get_trig_records(int, int *, int *)
report_error(int action_id,int error_code)

List of System Service Functions
Miscellaneous Functions:
set_flag(int flagnum, int flagval)
get_flag(int flagnum)
get_verbose_flag
get_abort_flag
set_led(int ledbit, int ledswitch)
writebit_LL_port(int portnum, int bitnum, int bitval)
write_LL_port(int portnum, int word)
readbit_LL_port(int portnum, int bitnum)
read_LL_port(int portnum, int portmask)
set_bit(int inword, int bitnum, int bitval)
get_bit(int inword, int bitnum)
t_clock
wait
wait_cycle(int numcycles)
int_max(int, int)
int_a_max(int, int *)

2.9 Description of System Service Functions, Alphabetical List

int assemble_header(int action_id, int messlength, int *mess_pointer)
Assembles message header for message given by *action_id*, stores message header at *mess_pointer*.

int exe_trig_calib(void)
Generates trigger accept and calibration pulse.

int export_message(int *pointer)
Exports message from DSP to VME-CPU, message stored at **pointer*. The function forces the export of the message. A VME-CPU message pending at that time is aborted prior to the export of the message. If the message to be exported is aborted by the VME-CPU, the message is lost and will not be sent again.

int get_abort_flag(void)
Returns the value of the abort flag. This function calls *get_flag*(ABORT_FLAG). If this call is not successful, it returns zero for the value of the abort flag.

int get_bit(int inword,int bitnum)
Returns bit *bitnum* of input word *inword*.

int get_flag(int flagnum)
Returns value of flag *flagnum* on success, -1 otherwise.

int get_trig_records(int trigblock, int *trigrec1, int *trigrec2)
Returns trigger records **trigrec1* and **trigrec2* for trigger block *trigblock*. Returns 1 if no trigger records available, 0 otherwise.

int get_verbose_flag(void)
Returns the value of the verbose flag. This function calls *get_flag*(VERBOSE_FLAG). If this call is not successful, it returns the previous value of the verbose flag.

int int_max(int in1, int in2)
Returns maximum of *in1* and *in2*.

int int_a_max(int n, int *inarr)
Returns maximum element of array *inarr*.

int read_LL_port(int portnum, int portmask)
Returns word read from LL port *portnum*. *portmask* masks the readable bits.

int readbit_LL_port(int portnum, int bitnum)
Returns value of bit *bitnum* from LL port *portnum*.

int report_error(int action_id, int error)
Reports an error to the VME-CPU. Exports error message given by *action_id*. Returns

one if export of message failed, zero on success.

int send_response(int response_code)
Exports the DSP message *response*. This function must only be called if the flag VERBOSE_FLAG is set to 1.

int set_bit(int inword, int bitnum, int bitval)
Sets bit *bitnum* in word *inword* to value *bitval* and returns modified word.

int set_flag(int flagnum, int flagval)
Sets flag *flagnum* to the value *flagval*.

int set_led(int ledbit, int ledswitch)
Sets LED *ledbit* to the value *ledswitch*.

int set_sys_defaults(int *trigval, int volts, int delay1, int delay2, int enable, int runmode)
Sets default values for run mode, trigger mode, alignment FIFO delays, and DAC's, and enables/disables readout.

int t_clock(void)
Reads timing timer.

int wait(void)
Waits a few clock cycles.

int wait_cycle(int numcycles)
Waits *numcycles* clock cycles (one internal clock cycle corresponds to 50 ns).

int write_LL_port(int portnum, int value)
Writes word *value* to LL port *portnum*.

int writebit_LL_port(int portnum, int bitnum, int value)
Writes *value* of bit *bitnum* to LL port *portnum*.

2.10 List of System Functions

Some system functions and messages invoke functions which have to be coded by the user. A compilation of these user functions is given in section 3 *User Code*.

Unless specified otherwise, a function is defined as *int function(void)*. The action executed by a function is usually implied by the function name. *Subfunctions* are functions called by *function*. *Type* indicates whether a subfunction is a detector independent system function (s) or a user function (u) specific to either pixel or strips readout.

List of System Functions		
List of System Functions Initiated by Messages		
function		
	subfunction	***type***
Initialization/Setup:		
disable_readout		
enable_readout		
get_dsp_status		
	export_message(int *)	*s*
initialize		
	set_led(int, int)	*s*
	report_error(int, int)	*s*
	set_flag(int, int)	*s*
	init	*s*
	set_sys_defaults(int *, int, int, int, int, int)	*s*
	det_init	*u*
	set_defaults	*u*
	init_user_var	*u*
load_graycode_pattern		
	report_error(int, int)	*s*
	store_gray	*s*
	load_gray	*s*
select_align_delay		
	report_error(int, int)	*s*
	set_align_delay(int delay1, int delay2)	*s*
select_run_mode		
	report_error(int, int)	*s*
select_time_param		
select_trigger_mode		
	set_trigger_mode(int *trig_pulses)	*s*
set_dac		
	report_error(int, int)	*s*
	set_voltage(int dac, int voltage)	*s*

List of System Functions		
List of System Functions Initiated by Messages		
function		
	subfunction	***type***
Data Collecting/Reporting:		
calibrate		
	calib_init(int user_par) set_burst_sig(int)	u s
load_data		
	report_error(int, int) load_data_block(int, int *)	s u
transfer_data		
	report_error(int, int) data_summary data_transfer(int data_format)	s u u
echo_reply		
	export_message(int *)	s
test_message		
	export_message(int *)	s

List of System Functions		
Message Handling:		
function		
	subfunction	***type***
handle_message		
	check_mess import_message execute_action report_error(int action_id, int error_code)	*s* *s* *s* *s*
execute_action		
	get_action_id dispatch_universal(int action_id) dispatch_specific(int action_id) report_error(int action_id, int error_code)	*s* *s* *u* *s*
dispatch_universal(int action_id)		
	get_verbose_flag(void) send_response(int)	*s* *s*
send_response(int response_code)		
	assemble_header(int, int, int *) export_message(int *)	*s* *s*
check_mess		
import_message		
export_message(int *pointer)		
	export_mess(int *pointer)	*s*
get_action_id		
assemble_header(int, int, int *)		
report_error(int action_id,int error_code)		
	export_message(int *pointer)	*s*

List of System Functions		
Setup and Initializations:		
function		
	subfunction	***type***
set_trigger_mode(int *)		
set_align_delay(int delay1, int delay2)		
set_voltage(int dac, int voltage)		
store_gray		
load_gray		
init		
	init_t_fifo init_a_fifo init_timer init_regs init_int_vecs init_mem	*s* *s* *s* *s* *s* *s*
burst_init		
	init_t_fifo init_a_fifo reset_timer(int, int) burst_det_init	*s* *s* *s* *u*
init_t_fifo		
init_a_fifo		
init_timer		
init_regs		
init_int_vecs		
init_mem		
reset_timer(int timernum, int clock)		
set_sys_defaults(int *, int, int, int, int, int)		
	set_trigger_mode(int *) set_voltage(int) set_align_delay(int, int) enable_readout/disable_readout run_mode = EOB_TRANSFER	*s* *s* *s* *s*
report_error(int action_id, int error_code)		
	export_message(int *pointer)	*s*

List of System Functions		
Data Collecting:		
readout_eob, readout_rt		
	data_ready read_det read_trigger: interrupt driven t_clock set_bit(int,int,int)	*u* *u* *s* *s* *s*
finish_readout		
	readout_eob or readout_rt data_transfer report_error(int, int) get_bit(int, int) wait_cycle(int)	*s* *s* *s* *s* *s*
exe_trig_calib		
read_trigger, interrupt driven		
burst_status(int b_sig)		
	get_flag(int)	*s*
set_burst_sig(int b_sig)		
	set_flag(int, int)	*s*
reset_burst_sig(int b_sig)		
	set_flag(int, int)	*s*
report_error(int action_id, int error_code)		
	export_message(int *pointer)	*s*
Data Reporting:		
data_transfer		
	assemble_trigdat assemble_header(int, int, int *) export_message(int *) report_data(int)	*s* *s* *s* *u*
data_summary		
	get_data_summary(int, int *, int *) export_message(int *)	*u* *s*
assemble_trigdat		
get_gray(int grayadd)		
get_trig_records(int, int *, int *)		
report_error(int action_id, int error_code)		
	export_message(int *pointer)	*s*

List of System Functions		
Miscellaneous Functions:		
function		
	subfunction	***type***
get_code_ver		
get_com_code_ver		
get_user_code_ver		
set_flag(int flagnum, int flagval)		
get_flag(int flagnum)		
get_verbose_flag(void)		
get_abort_flag(void)		
set_led(int ledbit, int ledswitch)		
writebit_port(int portnum, int bitnum, int bitval)		
write_port(int portnum, int word)		
readbit_port(int portnum, int bitnum)		
read_port(int portnum, int portmask)		
set_bit(int inword, int bitnum, int bitval)		
get_bit(int inword, int bitnum)		
t_clock		
wait		
wait_cycle(int numcycles)		
int_max(int, int)		
int_a_max(int, int *)		
List of System Utility Functions		
function		
	subfunction	***type***
writebit_LL_port(int portnum, int bitnum, int bitval)		
write_LL_port(int portnum, int word)		
readbit_LL_port(int portnum, int bitnum)		
read_LL_port(int portnum, int portmask)		

2.11 Description of System Functions, Alphabetical List

int assemble_header(int action_id, int messlength, int *mess_pointer)
Assembles message header for message given by *action_id*, stores message header at *mess_pointer*.

int assemble_trigdat()
Decodes trigger words read from trigger FIFO and from trigger counter and assembles trigger records one and two.

int burst_init(void)
Executes initializations necessary before a data taking cycle, calls user function *burst_det_init*.

int burst_status(int b_sig)
Returns status of *b_sig*. Start of burst signal: *b_sig* = 0, end of burst signal: *b_sig* = 1. Signal set: return value = 1. The return value is the logical 'OR' of the status of the hardware and the software burst signal.

int calibrate(void)
Analyzes message, sets run mode to CALIB_RUN, resets flag DSP_DATA_READY_FLAG, activates start of burst signal, and calls user function *calib_init(userpar)*.

int check_mess(void)
Checks whether message is pending. Returns the message length in 32-bit words if a message is pending, zero otherwise.

int data_transfer(int data_format)
Decodes trigger data and assembles trigger records. If *data_format* = 3, (trigger data en bloc), it assembles and exports the message data_transfer. If *data_format* = 1 or 2, it calls the user function *report_data(data_format)*.

int data_summary(void)
Calls the user function *get_data_summary(trig_blocks, &flag, &numblocks)* and assembles and exports the message data_summary.

int disable_readout(void)
Disables readout of detector.

int dispatch_universal(int action_id)
Executes action specified by *action_id* and sends message *response(response_code)* if flag VERBOSE_FLAG is set to 1 and no error message has been sent by action function.

int echo_reply(void)
Replies to message *echo*. Sends back message received.

int enable_readout(void)
Enables readout of detector.

int execute_action(void)
Analyzes imported message, calls *get_action_id* and dispatcher functions *dispatch_universal* and *dispatch_specific*.

int exe_trig_calib(void)
Generates trigger accept and calibration pulse.

int export_message(int *pointer)
Exports message from DSP to VME-CPU, message stored at **pointer*. The function forces the export of the message. A VME-CPU message pending at that time is aborted prior to the export of the message. If the message to be exported is aborted by the VME-CPU, the message is lost and will not be sent again.

int finish_readout(void)
Last readout cycle after end of burst (needed in case a trigger accept was received shortly before the DSP detected the end of burst signal).

int get_abort_flag(void)
Returns the value of the abort flag. This function calls *get_flag*(ABORT_FLAG). If this call is not successful, it returns zero for the value of the abort flag.

int get_action_id(void)
Analyzes message header and returns *action_id*.

int get_bit(int inword,int bitnum)
Returns bit *bitnum* of input word *inword*.

int get_com_code_ver(void)
Returns communication code version number.

int get_code_ver(void)
Returns system code version number.

int get_dsp_status(void)
Returns DSP setup parameters by sending message *dsp_status*.

int get_flag(int flagnum)
Returns value of flag *flagnum* on success, -1 otherwise.

int get_gray(int grayadd)
Returns graycode for graycode RAM address *grayadd*.

int get_trig_records(int trigblock, int *trigrec1, int *trigrec2)
Passes trigger records **trigrec1* and **trigrec2* for trigger block *trigblock*. Returns 1 if no

trigger records available, 0 otherwise.

int get_verbose_flag(void)
Returns the value of the verbose flag. This function calls *get_flag*(VERBOSE_FLAG). If this call is not successful, it returns the previous value of the verbose flag.

int handle_message(void)
Looks for incoming messages, calls *import_message* and *execute_action*.

int import_message(void)
Imports message from VME-CPU to DSP.

int init(void)
Executes DSP card initializations.

int init_a_fifo(void)
Initializes alignment FIFO's.

int init_mem(void)
Initializes memory blocks used for storing data (detector data and trigger data).

int init_regs(void)
Initializes DSP registers.

int init_t_fifo(void)
Initializes trigger FIFO.

int init_timer(void)
Initializes trigger timer (beam crossing counter) and timer used for timing.

int initialize(void)
Executes DSP card initializations. Calls *init*, *set_sys_defaults(trigval, dac_volts, al_delay1, al_delay2, enable, runmode)*, and user functions *det_init*, *set_defaults()*, and *init_user_var*. All flags, except VERBOSE_FLAG, are initialized to zero.

int int_max(int in1, int in2)
Returns maximum of *in1* and *in2*.

int int_a_max(int n, int *inarr)
Returns maximum element of array *inarr*.

int load_data(void)
Analyzes message. If the data format = 3 (trigger block), it stores the data in memory at p_trig_data. If the data format is 2 (detector data en bloc, link number given by message header word channel), it calls the user function *load_data_block(numdat, data_array)*. The function returns one if the error function *report_error*(action_id, error_code) has been called, zero otherwise.

int load_gray(void)
Loads graycode pattern into graycode RAM.

int load_graycode_pattern(void)
Stores graycode pattern in RAM (*store_gray*) and calls *load_gray*. The function returns 1 if the error function *report_error*(FATAL_ERROR, GRAYCODE_TOO_LONG) has been called.

int message_status(int stat_bit)
Returns status of message (message pending).

int read_port(int portnum, int portmask)
Returns word read from port *portnum*. *portmask* masks the readable bits.

int read_LL_port(int portnum, int portmask)
Returns word read from LL port *portnum*. *portmask* masks the readable bits.

int read_trigger(void)
Reads one word from trigger FIFO and reads trigger counter, stores data in memory. Returns an error flag.

int readbit_port(int portnum, int bitnum)
Returns value of bit *bitnum* from port *portnum*.

int readbit_LL_port(int portnum, int bitnum)
Returns value of bit *bitnum* from LL port *portnum*.

int readout_eob(void), int readout_rt(void)
Main readout function. Called by *main*. Readout of detector data (readout of trigger FIFO is interrupt driven). Calls user functions *data_ready* and *read_det*.

int report_error(int action_id, int error)
Reports an error to the VME-CPU. Exports error message given by *action_id*. Returns one if export failed, zero on success.

int reset_burst_sig(int b_sig)
Resets the software burst signal *b_sig*. Start of burst signal: $b\_sig = 0$; end of burst signal: $b\_sig = 1$. The software signals are only reset in a calibration run and in case of a fatal error during normal data taking (controlled by software signals). It is the VME-CPU's responsibility to reset these signals for software controlled data taking.

int reset_timer(int timernum, int clock)
Resets and restarts DSP timer *timernum*, sets clock mode (external or internal) given by *clock*. $clock = 0$: external clock, $clock = 1$: internal clock.

int select_align_delay(void)
Executes setting of trigger alignment FIFO delay. Analyzes message and calls

set_align_delay. The function returns 1 if the error function
report_error(FATAL_ERROR, INVALID_AL_DELAY) has been called.

int select_run_mode(void)
Analyzes message, sets runmode to mode given by message parameter. Run modes: end of burst data transfer mode, event by event data transfer mode, and calibration run. The function returns 1 if the error function *report_error*(FATAL_ERROR, INVALID_RUNMODE) has been called.

int select_time_param(void)
Analyzes message, sets timing constants given by message parameters.

int select_trigger_mode(void)
Analyzes message, calls *set_trigger_mode*.

int send_response(int response_code)
Exports the DSP message *response*. Called only if the flag VERBOSE_FLAG is set to 1.

int set_align_delay(int delay1, int delay2)
Programs alignment FIFO delays. FIFO 1 (trigger sources 0 and 1): *delay1*; FIFO 2 (trigger source 2): *delay2*. The delays are given in units of external clock cycles.

int set_bit(int inword, int bitnum, int bitval)
Sets bit *bitnum* in word *inword* to value *bitval* and returns modified word.

int set_burst_sig(int b_sig)
Software controlled setting of burst signal *b_sig*. Start of burst signal: $b\_sig = 0$, end of burst signal: $b\_sig = 1$. The DSP controls these signals only in a calibration run. It is the VME-CPU's responsibility to set and reset these signals for software controlled data taking.

int set_dac(void)
Executes DAC settings. Analyzes message and calls *set_voltage*. The function returns one if an error message has been sent, zero otherwise.

int set_flag(int flagnum, int flagval)
Sets flag *flagnum* to the value *flagval*.

int set_led(int ledbit, int ledswitch)
Sets LED *ledbit* to the value *ledswitch*.

int set_sys_defaults(int *trigval, int volts, int delay1, int delay2, int enable, int runmode)
Sets default values for run mode, trigger mode, alignment FIFO delays, and DAC's, and enables/disables readout.

int set_trigger_mode(int *trig_pulses)

Programs the mulpiple pulse generation logic for each trigger source. Programs the number of trigger accepts to generate per incoming trigger. The pointer **trig_pulses* points to the four dimensional array of values (determined by *select_trigger_mode*).

int set_voltage(int dac, int voltage)
Sets the output voltage of the DSP card DAC *dac* to *voltage*. *dac* = 0-3 : DAC's on DSP card, *voltage* = 0-4095 (4095 $\hat{=}$ 2.5 V).

int store_gray(void)
Stores graycode pattern in memory.

int test_message(void)
Function initiated by message *message_test*. Creates message according to the parameters given in *message_test* and sends the message to the VME-CPU.

int t_clock(void)
Reads timing timer.

int transfer_data(void)
Analyses message, determines data format. If the data format it negative, the function *data_summary* is called prior to calling the function *data_transfer*. The function returns 1 if the error function *report_error*(FATAL_ERROR, FORMAT_UNKNOWN) has been called.

int trigdat_ready(void)
Checks whether there is any data in the trigger FIFO, returns a flag. Return value = 1: trigger data ready.

int wait(void)
Waits a few clock cycles.

int wait_cycle(int numcycles)
Waits *numcycles* clock cycles (one internal clock cycle corresponds to 50 ns).

int write_port(int portnum, int value)
Writes word *value* to port *portnum*.

int write_LL_port(int portnum, int value)
Writes word *value* to LL port *portnum*.

int writebit_port(int portnum, int bitnum, int value)
Writes *value* of bit *bitnum* to port *portnum*.

int writebit_LL_port(int portnum, int bitnum, int value)
Writes *value* of bit *bitnum* to LL port *portnum*.

3 User Code

The system requires the user to assume the following responsibilities:

- provide the general user functions as required and specified by the system
- provide user specific functions
- conform to the organization of the memory
- maintain consistency with the overall system model

System resources are available for coding. These system service functions are described and listed in section 2.8.

User code is defined in the following files:

- *user.h*: definitions of user pointers and variables, structure definitions of user messages, definitions of the corresponding action_id's, definitions of user error and response codes, definitions of default values, definitions of general user function prototypes, and definitions of user-specific function prototypes.
- *user.c*: general user functions (**coding mandatory**) and user-specific functions. This file has to include *sys.h, sys_ext.h, and user.h*. *sys_util.h* has to be included if utility functions are called.

Skeletons of both of the files are provided.

3.1 Memory Organization

The user has access to two sections of the memory: the message buffer and the data buffer. The message buffer is used for temporary storing a message imported by the system and for assembling and storing a message before it is being exported. The message buffer is limited to MESS_BLOCKSIZE words (defined in *sys.h*). The globally defined pointer *p_mess* of type *struct m_generic* points to the beginning of the message buffer at MESS_BASE (defined in *sys.h*). The data section provides space for storing detector data and other user data. The globally defined pointer *const int *p_data* points to the beginning of the data buffer at DATA_BASE (defined in *sys.h*). The data buffer is limited to DATA_BLOCKSIZE words (defined in *sys.h*). For details of the organization of the data buffer, refer to section 3.3.

3.2 User Messages

3.2.1 User Messages VME → DSP, Dispatcher

The user can add messages to the messages already defined by the system. A user message has to be defined in *user.h*:

```
struct m_usermess {
  HEADER
  int user_par1;
  int user_par2;
  /* ... */
};
typedef struct m_usermess m_usermess;
```

The message header is defined in *sys.h*:

```
#define HEADER    unsigned int ciw; \
                  unsigned int len; \
                  unsigned int dsp; \
                  unsigned int dsp_dipsw; \
                  unsigned int channel; \
                  unsigned int host_ciw; \
                  unsigned int spare[5];
```

For each message, an action function has to be provided. The function prototype and the function's action_id have to be defined in *user.h* (strips: action_id 200-299, pixels: action_id 100-199):

```
#define USERMESS 200
int usermess(void);
```

The action function has to be called by the user function *dispatch_specific*:

```
int dispatch_specific(int action_id)
{
  int exeflag;
  int error;

  switch (action_id) {
  case USERMESS:
    error = usermess();
    if (! error) if (get_verbose_flag()) send_response(DONE);
    exeflag = 1;
    break;
  default:
    exeflag = 0;
  }
  return exeflag;
}
```

The function *send_response* must not be called if the action function *usermess* itself exports a message (including any error message) from the DSP to the VME. The function *dispatch_specific(int action_id)* is required by the system. If the user does not add any user messages, the function has to return the value 0.

The action function is responsible for extracting any parameters from the imported user message:

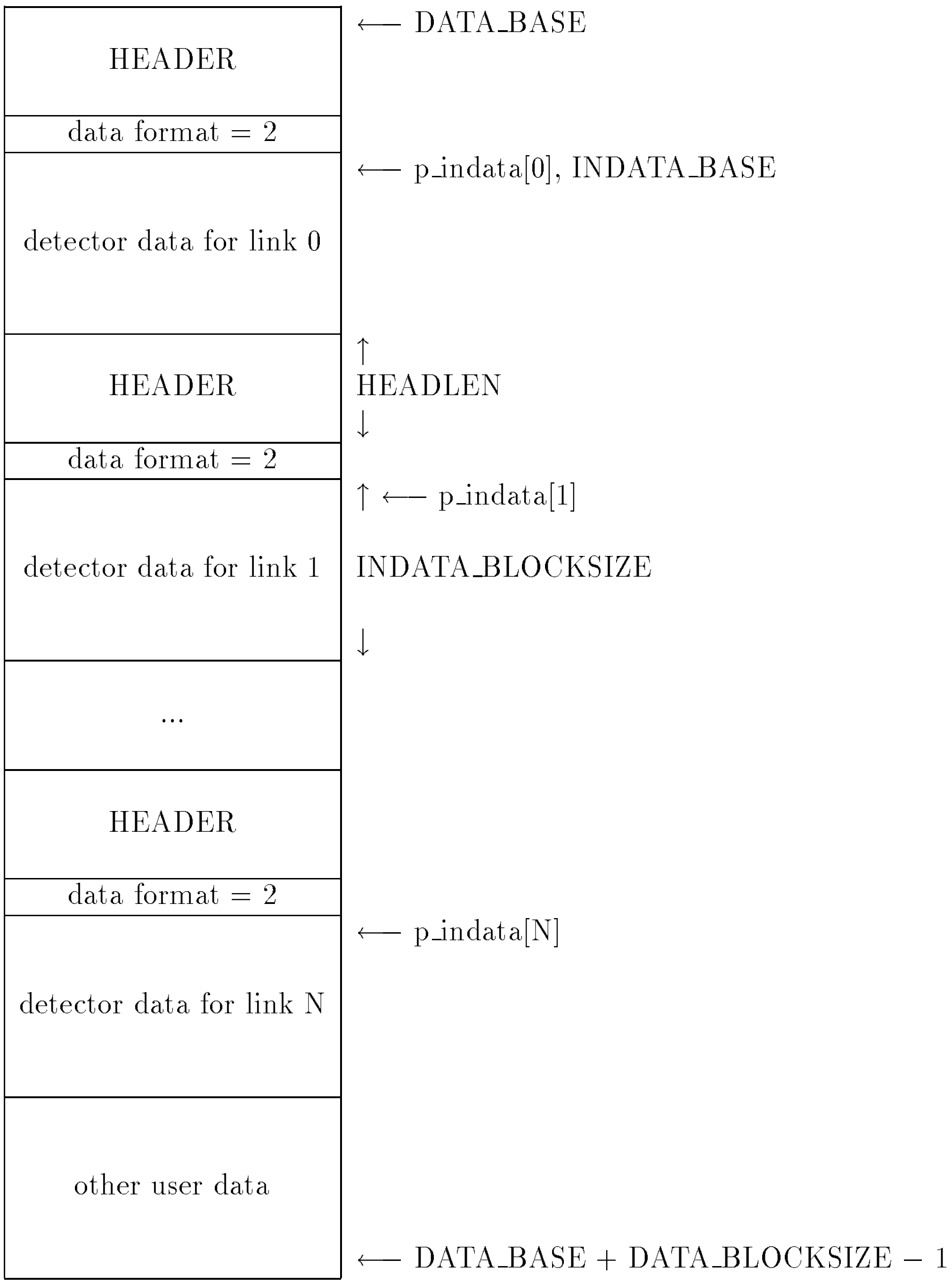
```
int usermess(void)
{
  int up1, up2;
  int error;
  m_usermess *mpt;

  mpt = (( m_usermess *) p_mess);
  up1 = mpt->user_par1;
  up2 = mpt->user_par2;
  /* ... */
  if (error) {
    report_error(FATAL_ERROR, MEM_OVERFLOW);
    return 1;
  }
  return 0;
}
```

3.2.2 User Messages DSP → VME

DSP → VME user messages can only be exported within action functions which are invoked by a VME → DSP user message. An *action_id* must be assigned to the message and should be defined in *user.h*. The message to be exported must be assembled in either the message buffer at *p_mess* or in the data buffer. The message header can be assembled by calling the system function *assemble_header(int action_id, int messlength, int *mess_pointer)*:
*assemble_header(action_id, messlength, (int *) p_mess)*;
The message parameters have to be stored beginning at *int *ploc = (int *) p_mess +* HEADLEN. The message is exported by calling *export_message((int *) p_mess)*. The function *export_message*(int *) forces the export of the message. A VME-CPU message pending at that time is aborted prior to the export of the message. If the message to be exported is aborted by the VME-CPU, the message is lost and will not be sent again.

3.3 Memory Organization, Data Section

The user should organize the data buffer in the following way:


HEADER
data format = 2
detector data for link 0
HEADER
data format = 2
detector data for link 1
...
HEADER
data format = 2
detector data for link N
other user data
⟵ DATA_BASE
⟵ p_indata[0], INDATA_BASE
↑
HEADLEN
↓
↑ ⟵ p_indata[1]
INDATA_BLOCKSIZE
↓
⟵ p_indata[N]
⟵ DATA_BASE + DATA_BLOCKSIZE − 1

It is recommended to define the following constants, pointers, and variables in *user.h*:

```
#define NUMLINKS 4                    /* number of links connected to DSP */
#define INDATA_BLOCKSIZE 0x0          /* max = DATA_BLOCKSIZE/NUMLINKS
                                         − (HEADLEN + 1) */
                                      /* (no other user data block) */
#define INDATA_BASE (DATA_BASE + HEADLEN + 1)

const int *p_indata[NUMLINKS] = {
  (int *) INDATA_BASE,
  (int *) (INDATA_BASE + HEADLEN + 1 + INDATA_BLOCKSIZE),
  (int *) (INDATA_BASE + 2*(HEADLEN + 1) + 2*INDATA_BLOCKSIZE),
  (int *) (INDATA_BASE + 3*(HEADLEN + 1) + 3*INDATA_BLOCKSIZE)};
```

p_indata[i] points to the input data block of link *i*.

```
static int indata_words[NUMLINKS];
static int indata_blocks[NUMLINKS];
```

indata_words[i] is the number of 32-bit words read and stored in memory for link *i*. *indata_blocks[i]* is the number of data blocks stored in memory for link *i* (data for 1 trigger accept corresponds to 1 data block).

3.4 Low Level Card (LL) Input and Output Ports

The system provides functions to write to LL output ports and to read from LL input ports. These functions and a structure *LL_ports* are defined in *sys_util.h* and *sys_util.c*:

```
struct LL_ports {
   volatile int *address;
   volatile int current_values;
};
```

In order to use these functions, the user has to define an array of ports in *user.h*.

Base addresses of LL ports (LL_BASE is defined in *sys_util.h*):

```
#define LL_IN_0_BASE          (LL_BASE + 0x0)
#define LL_IN_1_BASE          (LL_BASE + 0x0)
/* ... */
#define LL_OUT_0_BASE         (LL_BASE + 0x0)
#define LL_OUT_1_BASE         (LL_BASE + 0x0)
/* ... */
```

Definition of Ports (LL_MAXPORTS is defined in *sys_util.h*):

```
#define LL_IN0          0
#define LL_IN1          1
#define LL_OUT0         2
#define LL_OUT1         3
/* ... */

#define A_LL_IN0        (int *) LL_IN_0_BASE
#define A_LL_IN1        (int *) LL_IN_1_BASE
#define A_LL_OUT0       (int *) LL_OUT_0_BASE
#define A_LL_OUT1       (int *) LL_OUT_1_BASE
/* ... */

struct LL_ports LL_port[LL_MAXPORTS] = {
{A_LL_IN0, 0},
{A_LL_IN1, 0},
{A_LL_OUT0, 0},
{A_LL_OUT1, 0}};
/* ... */
```

Bit Definitions for LL Ports:

```
#define LL_IN0_BIT0      0
#define LL_IN0_BIT1      1
/* ... */
#define LL_OUT0_BIT0     0
#define LL_OUT0_BIT1     1
/* ... */
```

Input Port Masks:

```
#define M_LL_0      0xff      /* 8 bit input port */
#define M_LL_1      0xffff    /* 16 bit input port */
/* ... */
```

The following system utility functions can be used to write to and read from LL-ports:

- int writebit_LL_port(int portnum, int bitnum, int bitval)
- int write_LL_port(int portnum, int word)

- int readbit_LL_port(int portnum, int bitnum)
- int read_LL_port(int portnum, int portmask)

Example:

- writebit_LL_port(LL_OUT0, LL_OUT0_BIT1, 0);
- readword = read_LL_port(LL_IN1, M_LL_1);

3.5 Setup and Initializations

The user has to provide the following functions:

• int get_user_code_ver(void)
This user function is called by the system main function. The function has to return the version number of the user code. Byte 0 (LSB): version of *user.h*, byte 1: version of *user.c*.

• int det_init(void)
This user function is called by the system function *initialize*. *initialize* is executed before the beginning of the main program loop and upon receipt of the message *initialize*. The function *det_init* allows the user to initialize the detector.

• int set_defaults(void)
This user function is called by the system function *initialize*. It allows the user to set default values (which can be defined in *user.h*). The system sets default values for the trigger mode (one trigger accept per incoming trigger pulse), the trigger alignment FIFO delays (2 μs), the DAC's on the DSP card (0 V), the run mode (EOB_TRANSFER), and it enables the readout of the detector. These default values can be overriden by calling the function *set_sys_defaults(int *trigval, int volts, int delay1, int delay2, int enable, int runmode)*, where **trigval* points to the four dimensional array of values for the number of trigger accept pulses to be set for each of the four trigger sources.

• int init_user_var(void)
This user function is called by *initialize*. It allows the user to initialize user variables. The function should reset the data buffer word counts *indata_words*[NUMLINKS] and *indata_blocks*[NUMLINKS].

• int burst_det_init(void)
This user function is called by the system function *burst_init* which is executed before a data taking sequence. *burst_det_init* allows the user to initialize detector systems and user variables.

• int calib_init(int userpar)
This user function is called by the system function *calibrate*. *calibrate* is invoked upon receipt of the message *calibrate*. The function *calibrate* extracts the argument *userpar* of the user function from the message received. *calib_init* allows the user to perform initializations necessary before a calibration run. In case of an error, *calib_init* has to return a value greater than zero. Returning an error causes the cancellation of the calibration run.

3.6 Data Collecting

The system main function provides the framework for data collection (see section 1.4, page 8 and section 1.5, page 9). The user has to provide the following functions:

• int load_data_block(int numwords, int data[])
This user function is called by the system function *load_data* which is executed upon receipt of the message *load_data*. The VME message *load_data* transmitts data for one link. The user function has to load the data into the corresponding data buffer in memory. The number of data words in *data[]* is specified by *numwords*. The link number is part of the message header and is globally accessible through the variable *header_channel*. If the link number is negative, the data has to be loaded into the data buffer of each link. The variables *indata_words[i]* and *indata_blocks[il]* have to be set. The function has to return the value one if an error occurred and an error message has been sent (*e.g.* *report_error*(FATAL_ERROR, MEM_OVERFLOW) or *report_error*(FATAL_ERROR, INVALID_LINK)), zero otherwise.

• int exec_calib(void)
This user function is called by the system function *main* when the run mode is a calibration run. The function is called for each calibration event. The function has to execute any action necessary for calibration. For strips: load calibration control block into output FIFO and execute control block. For Pixels: generate calibration pulse and trigger accept (call system service function *int exe_trig_calib(void)*). The function *exec_calib*() has to return the value of the flag ABORT_FLAG which is set by the VME-CPU:
    return get_abort_flag();

• int data_ready(void)
This user function is called by the system functions *main* and *readout*. It has to return the value 1 when any detector data is ready to be read out.

• int read_det(void)
This is the main detector readout function. It is called by the system function *readout_eob* or *readout_rt*. The system readout function repeatedly calls the user function *read_det* as long as *data_ready* is true. The trigger information will be read upon receipt of an interrupt generated by the trigger FIFO, meaning that the function *read_det* does not have any time

constraint.

The function *read_det* has to store the data at the appropriate memory location in the data buffer. Note that there is no other memory space available than the data buffer. The data has to be stored in such a way that it can be transferred en bloc (data format 2) without copying/rearranging it. *read_det* has to return an error flag. No error: return 0, memory overflow: return MEM_OVERFLOW, other readout error: return RO_ERROR. MEM_OVERFLOW and RO_ERROR are defined in *sys.h*. Other error codes can be defined in *user.h* (see section 2.4 on pages 25ff).

3.7 Data Reporting

The system main function provides the framework for data reporting (see section 1.5, page 9). The user has to provide the following functions:

• int get_data_summary(int trigblocks, int *flag, int *blocks)
This function is called by the system function *data_summary*. The function *data_summary* is executed upon receipt of the VME message *transfer_data* if the data format requested is negative and the message *validate_data*. The function *get_data_summary* has to determine the summary flag **flag* and the number of data blocks **blocks*. One data block is defined as the data for one trigger accept. If the number of data blocks is equal to the number of trigger blocks *trigblocks*, the flag can be set to 0 and to 1 otherwise. If there is no trigger information available (no trigger control card), the calling function will set *trigblocks* to -1. However, it is the users's choice how the flag is defined and what the meaning of **blocks* is (average, maximum, ...) in case of more than one link.

• int report_data(int data_format)
This function is called by the system function *data_transfer*. The function *data_transfer* is executed upon the receipt of the VME message *transfer_data*. The function *report_data* is called for data transfers in formats 1 (1 message per trigger accept containing the trigger information and the data from all links for this trigger accept) and 2 (detector data from one link en bloc, 1 message total), and user defined data formats > 3.

Data format *data_format* = 1:
Each message has to be assembled in the message buffer at *(int *) p_mess*. The message header of size HEADLEN has to be followed by the parameter *data_format* at *ploc* = *(int *) p_mess* + HEADLEN , the message number, the two trigger records (set to 0 if no trigger information is available) and the data words. The system function *int get_trig_records(int trigblock, int *trigrec1, int *trigrec2)* can be called to get the two trigger records **trigrec1* and **trigrec2* for trigger accept *trigblock*. This system function returns 0 if trigger records were available, 1 otherwise: *nomore_trigs = get_trig_records(i, &trigrec1, &trigrec2)*. The system function *assemble_header(int action_id, int messlength, int *mess_pointer)* can be called to assemble the message header: *assemble_header*(DATA_TRANSFER,*messlength,*

*(int *) p_mess)*. Once the message is assembled it has to be exported by calling the system function *export_message((int *) p_mess)*. The procedure described above has to be repeated as many times as necessary to transmit all the data and trigger blocks. In RT_TRANSFER mode the user should call the function *init_user_var* after each data transfer to reset the data buffer word counts.

Data format *data_format* = 2:
The data from one link has to be transmitted in one message. The message has to be assembled in the data buffer. If the user organizes the data buffer as suggested in section 3.3 on pages 50ff, the user can write the following code to assemble and export the message:

```
messlength = indata_words[header_channel] + HEADLEN + 1;
ploc = (int *) p_indata[header_channel] − 1;
*ploc = data_format;
assemble_header(DATA_TRANSFER, messlength,
                (int *) p_indata[header_channel] − HEADLEN − 1);
export_message((int *) p_indata[header_channel] − HEADLEN − 1);
```

If the link number *header_channel* is negative, data for all links has to be transmitted in NUMLINKS messages *data_transfer*.

3.8 Flags, Error and Response Codes, LED's

The user can define flags accessible by the VME-CPU and the DSP in addition to the flags defined in the system code (see section 2.3). Flag numbers 0-20 are reserverd for system code, numbers 21-40 for pixel user code, and numbers 41-64 for strips user code. Flags and their values should be defined in *user.h*. The flags are accessible by the follwoing functions

- *int get_flag(int flagnum)*: returns the value of flag number *flagnum* on success, -1 otherwise.
- *int set_flag(int flagnum, int flagval)*: sets the flag number *flagnum* to the value *flagval*. Returns -1 for an invalid flag number, SUCCESS (0) otherwise.
- *int get_verbose_flag(void)*: returns the value of the verbose flag. This function calls *get_flag*(VERBOSE_FLAG). If this call is not successful, it returns the previous value of the verbose flag.
- *int get_abort_flag(void)*: returns the value of the abort flag. This function calls *get_flag*(ABORT_FLAG). If this call is not successful, it returns zero for the value of the abort flag.

Error and response codes can be defined in *user.h*, in addition to the codes defined by the system (see section 2.4). Error messages are sent by calling the function *int report_error(int action_id, int error_code)*. The parameter *action_id* is the action_id for the error message *fatal_error* or *error_report* (FATAL_ERROR or ERROR_REPORT). The messages *fatal_error* and *error_report* transmit the parameter *error_code* passed by the function *report_error*.

The DSP card has three LED's (green, red, yellow) which can be controlled by calling the function *set_led(int ledbit, int ledswitch)*:

- set_led(DSP_LED_G, DSP_LED_ON): turns on the green LED.
- set_led(DSP_LED_Y, DSP_LED_ON): turns on the yellow LED.
- set_led(DSP_LED_R, DSP_LED_OFF): turns off the red LED.

The LED's and the on/off switch are defined in *sys.h*.

3.9 List of General User Functions

Coding of these functions is mandatory. General user code is required by the system. The definitions of these functions are specified, but coding is the user's responsibility. The list of functions includes system functions which need to be invoked by the user functions. The function prototypes are defined in *user.h*. Unless specified otherwise, a function is defined as *int function(void)*. The action executed by a function is usually implied by the function name. *Subfunctions* are functions called by *function*. *Type* indicates whether a subfunction is a detector independent system function (s) or a user function (u) specific to either pixel or strips readout.

List of User Functions, Coding Mandatory		
function	***called by system function:***	
	subfunction	***type***
get_user_code_ver	main	
Dispatcher		
dispatch_specific(int action_id)	execute_action	
	get_verbose_flag(void) send_response(int)	*s* *s*
Setup and Initializations:		
det_init	initialize	
set_defaults	initialize	
calib_init(int userpar)	calibrate	
burst_det_init	burst_init	
init_user_var	initialize → init	
Data Collecting		
exec_calib	main	
	get_abort_flag(void)	*s*
load_data_block(int numwords, int data[])	load_data	
read_det	main → readout	
data_ready	main, readout	
Data Reporting		
report_data(int data_format)	data_transfer	
	assemble_header(int, int, int *) get_trig_records(int, int *, int *) export_message(int *) init_user_var	*s* *s* *s* *u*
get_data_summary(int, int *, int *)	data_summary	

4 Strips User Code

Strips user code refers to code controlling the UC Santa Cruz Low Level card. Strips user code conforms to the *User Code* defined in section 3. Strips user code is defined in the following files:

- *strips.h*: pointer and variable definitions, structure definitions of user messages, definitions of the corresponding action_id's, definitions of default values, definitions of general user function prototypes, and definitions of user-specific function prototypes.
- *strips.c*: general user functions (**coding mandatory**, see section 3.9) and user-specific functions.
- *stripdaq.h*: header file for *stripdaq.c*
- *stripdaq.c*: time critical detector readout functions. This part of the code is loaded into the DSP's internal RAM.

4.1 Organization of Data Buffer in Memory

The data buffer is organized following the principle described in section 3.3 of *User Code*. The pointers are defined as described in that section. In addition to the four blocks reserved for detector data, there are four blocks to be used to store output FIFO control blocks. The control block size CTROUT_BLOCKSIZE is (2048 + 1), 2048 being the maximum number of words to be loaded into the output FIFO. The first word of a control block in memory is the number of control words in the block to follow. The following pointers are defined to address these blocks:

```
const int *p_ctrout_block[4] = {
  (int *) CTROUT_BASE,
  (int *) (CTROUT_BASE + CTROUT_BLOCKSIZE),
  (int *) (CTROUT_BASE + 2*CTROUT_BLOCKSIZE),
  (int *) (CTROUT_BASE + 3*CTROUT_BLOCKSIZE)};
```

p_ctrout_block[i] points to the control block *i*.

```
#define INDATA_BLOCKSIZE                (DATA_BLOCKSIZE/NUMLINKS \
                                               - (HEADLEN + 1) \
                                         - 4*CTROUT_BLOCKSIZE)
#define CTROUT_BASE \
        (INDATA_BASE + 4*(HEADLEN + 1 + INDATA_BLOCKSIZE) + 0x0)
```

INDATA_BLOCKSIZE is 50'672 32-bit words.

HEADER	⟵ DATA_BASE
data format = 2	
detector data for link 0	⟵ p_indata[0], INDATA_BASE
...	
HEADER	↑ HEADLEN ↓
data format = 2	
detector data for link 3	↑ ⟵ p_indata[3] INDATA_BLOCKSIZE ↓
output FIFO control block 0	⟵ p_ctrout_block[0], CTROUT_BASE
...	
output FIFO control block 2	↑ CTROUT_BLOCKSIZE ↓
output FIFO control block 3	⟵ p_ctrout_block[3] ⟵ DATA_BASE + DATA_BLOCKSIZE − 1

4.2 List of Strips Messages

Strips, List of Messages: VME → DSP			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
load_control_block	m_load_control	2049	load output FIFO control block into RAM block [0-3]
		blocknum	RAM block
		blockdata[2048]	control block words (max 2048)
execute_control_block	m_exe_control	2	loads control block from RAM into output FIFO, executes block, this is repeated numexe times; no data is read out (send message 'calibrate' to activate readout).
		blocknum	RAM buffer block to load and execute
		numexe	number of repetitions (load, execute), if < 0: infinite loop
load_exe_ctr_blk	m_load_exe_ctr_blk	2051	loads control block given by blockdata and executes it once
		take_data	$= 1$: detector is read out and data transferred $= 0$: no readout
		data_format	format for data_transfer
		numpackets	number of packets expected
		blockdata[2048]	FIFO control block
get_det_status	m_get_det_status	2048	loads and executes FIFO control block, HAC/CDP128 status is sent by message det_status (use load_exe_ctr_block instead)
		blockdata[2048]	FIFO control block
dis_enable_link	m_dis_enable_link	1	enables or disables readout of specific link (given by message header field *channel*
		link_enable	> 0: enable link < 0: disable link

Strips, List of Messages: VME → DSP			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
get_LL_status	none	none	LL setup parameters
reset_fifos	none	none	resets input and output FIFO's
set_counter_val	m_set_counter	1	sets input FIFO control counter
		count_val	counter value, units of [bytes], 255 max
set_LL_dac	m_set_LL_dac	2	sets SLL-card DAC
		dac voltage	destination dac (0-3) voltage (0-4095)
set_strobe_delay	m_set_strobe	1	sets strobe delay
		strobe_delay	delay [0-255]
set_timeout	m_set_timeout	2	sets timeout
		det_rd_timeout	
		gen_timeout	

Strips, List of Messages: DSP → VME			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
det_status	m_det_status	2048	response to message get_det_status
		data[2048]	status block
LL_status	m_LL_status	12	response to get_LL_status
		LL_dac_setting[4]	
		LL_strobe_setting	
		LL_counter_setting	
		LL_timeout_setting[2]	
		LL_link_enabling[4]	

DSP responses:

- get_det_status ⇒ DSP message *det_status*
- get_LL_status ⇒ DSP message *LL_status*
- execute_control_block ⇒ ACCEPTED (if flag VERBOSE_FLAG = 1)
- load_exe_ctr_block ⇒ message(s) data_transfer if take_data = 1, DONE otherwise (if flag VERBOSE_FLAG = 1)
- all other messages ⇒ DONE (if flag VERBOSE_FLAG = 1)

4.3 Definitions of action_id

Numbers 100-199 are reserved for pixel-code action_id's, numbers 200-299 for strips-code action_id's, and numbers 1-99 for system-code action_id's.

Definitions of action_id	
Messages/Actions initiated by VME CPU	
Definiton	***Value***
DIS_ENABLE_LINK	212
EXECUTE_CONTROL_BLOCK	201
GET_DET_STATUS	202
GET_LL_STATUS	207
LOAD_CONTROL_BLOCK	200
LOAD_EXE_CTR_BLK	209
RESET_FIFOS	211
SET_STROBE_DELAY	204
SET_COUNTER_VAL	205
SET_LL_DAC	206
SET_TIMEOUT	210
Messages sent by DSP to VME	
DET_STATUS	203
LL_STATUS	208

4.4 Definitions of Error and Response Codes

Error codes are passed by the DSP messages *fatal_error* and *error_report*, response codes by the DSP message *response*. Numbers 1-99 are reserved for system codes (see page 25), number 100-199 for pixel-code, and numbers 200-299 for strips-code.

Definitions of Error and Response Codes		
Definiton	***Value***	***Error/Response***
	Fatal Errors	
BLOCK_TOO_LONG	200	output FIFO control block too long
INVALID_BLOCK_NUMBER	201	invalid output FIFO control block
	Other Errors	
INVALID_COUNTER_VALUE	210	input counter value out of range
INVALID_DELAY_VALUE	211	strobe delay value out of range
TIMEOUT	212	timeout
TOO_FEW_PACKETS	213	less packets received than expected
	Response Codes	

4.5 Strips Default Settings

Strips and system default values are set after the DSP code has been downloaded and by the message *initialize*.

Default Settings		
Message	***Parameter***	***Default Value***
select_run_mode	runmode	EOB_TRANSFER
enable/disable_readout	readout_enabled	1
select_trigger_mode	trignum[4]	4*1
select_align_delay	delay1 [clocks]	80
	delay2 [clocks]	80
dis_enable_link	link_enable	4*1
set_dac	voltage	4*0
set_strobe_delay	strobe_delay	0
set_counter_val	count_val	99
set_LL_dac	voltage	4*0
set_timeout [μs]	det_rd_timeout	1000
	gen_timeout	0

4.6 Data Format

The trigger information for each trigger is packed into two 32-bit words (see page 14). The corresponding strips data is encoded in ? 32-bit words per link.

The format of the data transmitted by messages *data_transfer* is selected by the parameter *data_format* which is submitted by the message *transfer_data* (see section 1.5) and *load_exe_ctr_blk*. The formats supported are listed below.

$data\_format = 1$:

Message # 1 = Event 1
Trigger Record 1 1
Trigger Record 1 2
# of Words for Link 0 (= N)
Link 0: Strips Data Word 1 1
...
Link 0: Strips Data Word 1 N
...
# of Words for Link 3 (= N)
Link 3: Strips Data Word 1 1
...
Link 3: Strips Data Word 1 N

...

...

Message # N = Event N
Trigger Record N 1
Trigger Record N 2
# of Words for Link 0 (= M)
Link 0: Strips Data Word N 1
...
Link 0: Strips Data Word N M
...
# of Words for Link 3 (= M)
Link 3: Strips Data Word N 1
...
Link 3: Strips Data Word N M

The information for each trigger is packed into one single message *data_transfer*. All messages, as many as there are triggers, are sent to the VME automatically once the DSP has received the request to transfer the data. The message number (normally corresponding to the event number) is encoded in the message as the second parameter (the first parameter is the data format).

$data\_format = 2$:

# of Data Words for Link i (all Triggers)
Link i: Strips Data Word 1 1
...
Link i: Strips Data Word 1 K
...
Link i: Strips Data Word N 1
...
Link i: Strips Data Word N M

The link for which the data will be transferred is specified by the word *channel* in the message header of the VME message *transfer_data* or *load_exe_ctr_blk*. If $channel < 0$: NUMLINKS messages.

$data\_format = 3$:

Trigger Record 1 1
Trigger Record 1 2
...
Trigger Record N 1
Trigger Record N 2

$data\_format = 4$:

Message # i = Packet i
# of Words for Link j (= N)
Link j: Strips Data Word i 1
...
Link j: Strips Data Word i N

Data in the formats 2 (if $channel \geq 0$) and 3 is transmitted in a single message *data_transfer*. Data in format 4 is transmitted in as many messages as needed to transfer all packets received by one link.

The *data_format* requested is encoded in the first parameter of each message *data_transfer*. This parameter is followed by the data. If the data format is negative, the DSP sends the message *data_summary* prior to transmitting the data.

Special Cases:

No trigger information (no trigger control logic on DSP):

data_format = 1:

Message # i
Trigger Record i 1 = 0
Trigger Record i 2 = 0
# of Words for Link 0 (= K)
Link 0: Strips Data Word i 1
...
Link 0: Strips Data Word i K
...
# of Words for Link 3 (= M)
Link 3: Strips Data Word i 1
...
Link 3: Strips Data Word i M

No Strips data from link j:

data_format = 1:

Message # i
Trigger Record i 1
Trigger Record i 2
# of Words for Link 0 (= K)
Link 0: Strips Data Word i 1
...
Link 0: Strips Data Word i K
...
# of Words for Link 2 = 0
# of Words for Link 3 (= M)
Link 3: Strips Data Word i 1
...
Link 3: Strips Data Word i M

4.7 Strips Low Level Card (SLL) Input and Output Ports

The SLL ports can be accessed by the functions

- int writebit_LL_port(int portnum, int bitnum, int bitval)
- int write_LL_port(int portnum, int word)
- int readbit_LL_port(int portnum, int bitnum)
- int read_LL_port(int portnum, int portmask)

The port numbers *portnum*, port masks *portmask*, and the port addresses are defined in *strips.h*. The array *LL_port*[LL_MAXPORTS] is initialized in *strips.h*. There is one port mask for all SLL ports: M_SLL (0xff). For more details see section B.3 (strips.h). The

Port	***Port number***	***Port Address***
Input FIFO Flags	SLL_INFLAG	A_SLL_INFLAG
Output FIFO Flags	SLL_OUTFLAG	A_SLL_OUTFLAG
Input FIFO Link 0	SLL_DATIN0	A_SLL_DATIN0
Input FIFO Link 1	SLL_DATIN1	A_SLL_DATIN1
Input FIFO Link 2	SLL_DATIN2	A_SLL_DATIN2
Input FIFO Link 3	SLL_DATIN3	A_SLL_DATIN3
DAC Control, FIFO Reset and GO	SLL_DACMISC	A_SLL_DACMISC
Input Control Counter	SLL_COUNTER	A_SLL_COUNTER
Output FIFO	SLL_DATOUT	A_SLL_DATOUT
Output Control Strobe Delay	SLL_DELAY	A_SLL_DELAY
Input FIFO 0 Readout Control	SLL_NEXT0	A_SLL_NEXT0
Input FIFO 1 Readout Control	SLL_NEXT1	A_SLL_NEXT1
Input FIFO 2 Readout Control	SLL_NEXT2	A_SLL_NEXT2
Input FIFO 3 Readout Control	SLL_NEXT3	A_SLL_NEXT3

following bits are defined:

SLL_OFIFO_RESET	0	FIFO Reset
SLL_OFIFO_GO	1	GO\ low: Output FIFO read out and data sent to links
SLL_OUTFLAG0	0	Output FIFO flag
SLL_OUTFLAG1	1	Output FIFO flag

The SLL-DAC control bits are defined in the same way as the DSP on-board DAC control bits (see page 29).

4.8 List of Strips User Functions

Unless specified otherwise, a function is defined as *int function(void)*. The action executed by a function is usually implied by the function name. *Subfunctions* are functions called by *function*. *Type* indicates whether a subfunction is a detector independent system function (s) or a user function (u) specific to either pixel or strips readout. **Coding of functions called by system functions is mandatory** (see page 59). These functions are listed in *italic*.

List of Strips User Functions		
List of Functions Initiated by User Messages		
function		
	subfunction	***type***
load_control_block		
	report_error(int, int)	*s*
	load_ctr_block(int *pload, int blocklength, int datawords)	*u*
load_exe_ctr_blk		
	init_user_var	*u*
	clear_indata_buf	*u*
	write_LL_port	*s*
	writebit_LL_port	*s*
	readbit_LL_port	*s*
	read_packets(int packets_expected)	*u*
	report_data(int data_format)	*u*
execute_control_block		
	report_error(int, int)	*s*
	exe_control(int blocknum, int numexe)	*u*
dis_enable_link		
	enable_link(int enable, int link)	*u*

List of Strips User Functions		
List of Functions Initiated by User Messages		
function		
	subfunction	***type***
get_det_status		
	report_error(int, int) init_user_var write_LL_port(int, int) writebit_LL_port(int, int, int) readbit_LL_port(int, int) wait data_ready read_det assemble_header(int, int, int *) export_message(int *)	*s* *u* *s* *s* *s* *s* *u* *u* *s* *s*
get_LL_status		
	export_message(int *)	*s*
set_strobe_delay		
	report_error(int, int) set_strobe(int delay)	*s* *u*
set_counter_val		
	report_error(int, int) set_counter(int counterval)	*s* *u*
set_LL_dac		
	report_error(int, int) set_LL_voltage(int dac, int voltage)	*s* *u*

List of Strips User Functions		
function		
	subfunction	***type***
get_user_code_ver		
Dispatcher		
dispatch_specific(int action_id)		
	get_verbose_flag(void) send_response(int)	*s* *s*
Setup and Initializations		
det_init		
	enable_link(int, int) reset_io_fifo	*u* *u*
set_defaults		
	set_sys_defaults(int *, int, int, int, int, int) set_LL_voltage(int, int) set_strobe(int) set_counter(int)	*s* *s* *u* *u*
calib_init(int userpar)		
	report_error(int, int) calib_block = userpar	*s*
burst_det_init		
	init_user_var reset_io_fifo clear_indata_buf	*u* *u* *u*
init_user_var		
set_strobe(int)		
	write_LL_port(int portnum, int value)	*s*
set_counter(int)		
	write_LL_port(int portnum, int value)	*s*
set_LL_voltage(int, int)		
	write_LL_port(int portnum, int value)	*s*
reset_io_fifo		
	writebit_LL_port(int, int, int)	*s*
clear_indata_buf		
enable_link(int, int)		

List of Strips User Functions		
function		
	subfunction	***type***
Data Collecting		
exec_calib		
	exe_control(calib_block, 1) get_abort_flag(void)	*u* *s*
load_data_block(int numwords, int data[])		
	report_error(int, int)	*s*
read_det		
	read_byte(int, int *)	*u*
read_byte(int linknum, int *byteval)		
	read_LL_port(int, int) write_LL_port(int, int)	*s* *s*
read_packets(int packets_expected)		
	t_clock read_LL_port(int, int) read_byte(int, int *) report_error(int, int)	*s* *s* *u* *s*
data_ready		
	read_LL_port(int, int)	*s*
exe_control(int, int)		
	get_abort_flag(void) write_LL_port(int, int) writebit_LL_port(int, int, int)	*s* *s* *s*
load_ctr_block(int *, int, int)		
Data Reporting		
report_data(int data_format)		
	assemble_header(int, int, int *) get_trig_records(int, int *, int *) export_message(int *) init_user_var	*s* *s* *s* *u*
*get_data_summary(int, int *, int *)*		

4.9 Description of Strips User Functions, Alphabetical List

int burst_det_init(void)
Executes initializations necessary before every new burst data taking cycle. Calls user function *init_user_var*.

int calib_init(int userpar)
Sets calibration block *calib_block* to execute to *userpar*. Returns 1 if *report_error*(FATAL_ERROR,INVALID_BLOCK_NUMBER) has been called, 0 otherwise.

int clear_indata_buf(void)
Clears data buffer from *p_data* to *p_ctrout_block*[0].

int data_ready(void)
Checks whether data is ready to be read from any of the input FIFO's. Returns ready-flag.

int det_init(void)
Detector initializations.

int dis_enable_link(void)
Analyzes message and calls *enable_link(enable, link)*.

int dispatch_specific(int action_id)
Called by *execute_action*. Executes action specified by *action_id* and calls function *send_response(response_code)* if flag VERBOSE_FLAG is set to 1. Returns 1 if action executed, 0 otherwise.

int enable_link(int enable, int link)
Enables ($enable > 0$) or disables ($enable = 0$) link number *link* for readout of data.

int exec_calib(void)
Calibration function, called for each calibration event. Calls function *exe_control(calib_block,1)* to load and execute control block once and returns value of flag ABORT_FLAG.

int exe_control(int blocknum, int numexe)
Assumes that control block stored in memory is not packed (i.e. each 32-bit word contains one 8-bit control word (LSB)). Loads control block *blocknum* from memory into output FIFO and executes control block. This sequence is repeated *numexe* times ($numexe < 0$: infinite loop) unless the abort flag assumes a value greater than zero. No data is read out.

int exe_control_packed(int blocknum, int numexe)
Assumes that control block stored in memory is packed (i.e. each 32-bit word contains four 8-bit control words). Loads control block *blocknum* from memory into output FIFO and executes control block. This sequence is repeated *numexe* times unless the abort flag assumes a value greater than zero. No data is read out.

int execute_control_block(void)
Analizes imported message and calls *exe_control* (or *exe_control_packed* depending on what was agreed on (no choice!)). As of June 22, 1995, the function calls *exe_control*. Returns 1 if
report_error(FATAL_ERROR, INVALID_BLOCK_NUMBER) has been called, 0 otherwise.

int get_data_summary(int trigblocks, int *flag, int *blocks)
Determines the number of data blocks **blocks*, compares it with the number of trigger blocks and sets **flag* accordingly. The 32-bit word *flag can be split up into 4 byte flags to accomodate the information of the 4 links. What should be returned as *blocks* has to be determined (average of four links, maximum of four links ... ?).

int get_det_status(void)
Loads control block passed by message *get_det_status* into output FIFO and executes it. The input FIFO's are then read out and the message *det_status* is sent to the VME. The function assumes that each 32-bit control block word contains one 8-bit word (LSB) to be loaded into the FIFO.

int get_LL_status(void)
Returns LL setup parameters (no readback of hardware).

int get_user_code_ver(void)
This user function is called by the system main function. The function has to return the version number of the user code. Byte 0 (LSB): version of *strips.h*, byte 1: version of *strips.c*.

int init_user_var(void)
Initializes variables used for readout.

int load_control_block(void)
Analyzes imported message. Determines memory block [0-3] for storage of control block and calls *load_ctr_block*. Returns 1 if
report_error(FATAL_ERROR, INVALID_BLOCK_NUMBER) or
report_error(FATAL_ERROR, BLOCK_TOO_LONG) has been called, 0 otherwise.

int load_exe_ctr_blk(void)
Loads control block from message load_exe_ctr_blk into output fifo, executes control block, and reads out input FIFO(s) if *take_data* is true. If the number of packets received within the time period given by the timeout setting *det_read_timeout* does not correspond to the number of packets expected the error message *error_report* is sent with the error code TOO_FEW_PACKETS. If *take_data* is true, the data will be transferred by the message(s) *data_transfer*.

int load_ctr_block(int *pload, int numwords, int *blockdata)
Loads control block given by the array *blockdata[numwords]* into memory at **pload*.

int load_data_block(int numwords, int data[])
Loads data in format 2 (data for link i en bloc) into memory. If the link number is less than zero all NUMLINKS data buffers are loaded. Returns 1 if *report_error*(FATAL_ERROR,MEM_OVERFLOW) or *report_error*(FATAL_ERROR,INVALID_LINK) has been called, 0 otherwise.

int read_byte(int linknum, int *byteval)
Reads one byte from input FIFO of link *linknum* and returns the value in *byteval*. Returns flag indicating whether byte was read.

int read_det(void)
Main detector readout function, called by *main* → *readout_eob* and *readout_rt*. Reads out input FIFO's and stores the data in the data buffers. The first byte of a data packet is always stored as the most significant byte of a 32-bit word. If the number of bytes in a data packet is not a multiple of four, the last word is filled up with zero's.

int read_packets(int packets_expected)
Called by *load_exe_ctr_blk()*.

int report_data(int data_format)
Called by *data_transfer*. Assembles data in format *data_format* = 1, 2, or 4 and exports message(s) DATA_TRANSFER.

int reset_io_fifo(void)
Resets output and input FIFO's.

int set_counter(int counterval)
Sets counting value of input FIFO control counter to *counterval* .

int set_counter_val(void)
Analyzes imported message and calls *set_counter*. Returns 1 if *report_error*(ERROR_REPORT,INVALID_COUNTER_VALUE) has been called, 0 otherwise.

int set_defaults(void)
Called by *initialize*. Sets user default values. Calls system function *set_sys_defaults* and user functions *set_LL_voltage*, *set_strobe*, and *set_counter*.

int set_LL_dac(void)
Analyzes message and calls *set_LL_voltage(dac, voltage)*. Returns 1 if *report_error*(ERROR_REPORT,INVALID_DESTINATION_DAC) or *report_error*(ERROR_REPORT,INVALID_DAC_VALUE) has been called, 0 otherwise.

int set_LL_voltage(int dac, int voltage)
Sets LL-card DAC *dac* to *voltage*.

int set_strobe(int delay)
Sets strobe delay to *delay*.

int set_strobe_delay(void)
Analyzes imported message and calls *set_strobe*. Returns 1 if *report_error*(ERROR_REPORT,INVALID_DELAY_VALUE) has been called, 0 otherwise.

5 Pixel User Code

Pixel user code refers to code controlling the LBL pixel Low Level card. Pixel user code conforms to the *User Code* defined in section 3. Pixel user code is defined in the following files:

- *pixels.h*: pointers and variables definitions, structure definitions of user messages, definitions of the corresponding action_id's, definitions of default values, definitions of general user function prototypes, and definitions of user-specific function prototypes.
- *pixels.c*: general user functions (**coding mandatory**, see section 3.9) and user-specific functions.

5.1 Organization of Data Buffer in Memory

The data buffer is organized following the principle described in section 3.3 of *User Code*. The pointers are defined as described in that section. Pixel code specific definitions:

#define NUMLINKS 1
#define INDATA_BLOCKSIZE (DATA_BLOCKSIZE − (HEADLEN + 1))
INDATA_BLOCKSIZE is 235'508 32-bit words.

5.2 List of Messages

Pixels, List of Messages: VME → DSP			
action_id = COMMAND			
command	***structure***	***parameters***	***function***
select_readout_mode	m_sel_read	1	selects PAC readout mode
		read_mode	readout mode 0: sparse 1: no skipping
load_masks	m_load_masks	25	loads masks
		mask_mode	0 = calibration masks 1 = disable masks
		mask[12][2]	pixel masks mask[0][0]: mask for upper 32 pixels of column 0 (pixels 32 - 63) mask[0][1]: mask for lower 32 pixels of column 0 (pixels 0 - 31) mask: 32-bit integer MSb pixel 63 or 31 LSb pixel 32 or 0
select_data_mode	m_sel_data_mode	1	selects 32- or 52-bit readout of ROC
		data_mode	0: 32 bits of ROC FIFO word read out 1: 52 bits of ROC FIFO word read out
reset_pac	m_reset_pac	none	global reset of pixel array

DSP response: DONE (if flag VERBOSE_FLAG = 1)

5.3 Definitions of action_id

Numbers 100-199 are reserved for pixel-code action_id's, numbers 200-299 for strips-code action_id's, and numbers 1-99 for system-code action_id's.

Definitions of action_id	
Messages/Actions initiated by VME CPU	
Definiton	***Value***
LOAD_MASKS	101
SELECT_DATA_MODE	103
SELECT_READOUT_MODE	100
RESET_PAC	102

5.4 Definitions of Error and Response Codes

Error codes are passed by the DSP messages *fatal_error* and *error_report*, response codes by the DSP message *response*. Numbers 1-99 are reserved for system codes (see page 25), number 100-199 for pixel-code, and numbers 200-299 for strips-code.

Definitions of Error and Response Codes		
Definiton	***Value***	***Error/Response***
Fatal Errors		
INVALID_READMODE	100	invalid PAC readout mode
INVALID_MASKMODE	101	invalid mask mode
INVALID_DATAMODE	102	invalid data mode
Other Errors		
Response Codes		

5.5 Pixels Default Settings

Pixels and system default values are set after the DSP code has been downloaded and by the message *initialize*.

Default Settings		
Message	***Parameter***	***Default Value***
select_run_mode	runmode	EOB_TRANSFER
enable/disable_readout	readout_enabled	1
select_trigger_mode	trignum[4]	4*1
select_align_delay	delay1 [clocks]	80
	delay2 [clocks]	80
select_readout_mode	read_mode	0
select_data_mode	data_mode	0
set_dac	voltage	4*0

5.6 Data Format

The trigger information for each trigger is packed into two 32-bit words (see page 14). The corresponding pixel-array-chip (PAC) data is encoded in one 32-bit word per pixel (32-bit readout mode):

PAC Data Word

31	30	29 26	25 23	22 20	19 16	15 8	7 0
Data Valid	TMark	TrigTag	Row A	Row B	Column	ROtimestamp	ADC Data

An additional 32-bit word is needed for the 52-bit readout mode. PAC FIFO bits 32-51 are stored in bits 0 - 19.

The format of the data transmitted by messages *data_transfer* is selected by the parameter *data_format* which is submitted by the message *transfer_data* (see section 1.5). The formats supported are listed below.

data_format = 1:

Message # 1 = Event 1
Trigger Record 1 1
Trigger Record 1 2
# of PAC Data Words (= K)
PAC Data Word 1 1
...
PAC Data Word 1 K

...
...

Message # N = Event N
Trigger Record N 1
Trigger Record N 2
# of PAC Data Words (= M)
PAC Data Word N 1
...
PAC Data Word N M

The information for each trigger is packed into one single message *data_transfer*. All messages, as many as there are triggers, are sent to the VME automatically once the DSP has received the request to transfer the data. The message number (normally corresponding to the event number) is encoded in the message as the second parameter (the first parameter is the data format).

data_format = 2:

PAC Data Word 1 1
...
PAC Data Word 1 M
...
...
PAC Data Word N 1
...
PAC Data Word N K

data_format = 3:

Trigger Record 1 1
Trigger Record 1 2
...
Trigger Record N 1
Trigger Record N 2

Data in the formats 2 or 3 is transmitted in a single message *data_transfer*.

The *data_format* requested is encoded in the first parameter of each message *data_transfer*. This parameter is followed by the data. If the data format is negative, the DSP sends the message *data_summary* prior to transmitting the data.

Special Cases:

No trigger information (no trigger control logic on DSP):

data_format = 1:

Message # i
Trigger Record i 1 = 0
Trigger Record i 2 = 0
# of PAC Data Words (= K)
PAC Data Word i 1
...
PAC Data Word i K

No PAC data for specific trigger i:

data_format = 1:

Message # i
Trigger Record i 1
Trigger Record i 2
# of PAC Data Words = 0

data_format = 2:

PAC Data Word i-1 1
...
PAC Data Word i-1 M
Invalid PAC Data Word i
PAC Data Word i+1 1
...
PAC Data Word i+1 K

No trigger information and no PAC data:

data_format = 1:

Message # i
Trigger Record i 1 = 0
Trigger Record i 2 = 0
# of PAC Data Words i 1 = 0

data_format = 2 and *data_format* = 3:

Message Header only

5.7 Pixels Low Level Card (PLL) Input and Output Ports

The PLL ports can be accessed by the functions

- int writebit_LL_port(int portnum, int bitnum, int bitval)
- int write_LL_port(int portnum, int word)
- int readbit_LL_port(int portnum, int bitnum)
- int read_LL_port(int portnum, int portmask)

The port numbers *portnum*, port masks *portmask*, and the port addresses are defined in *pixels.h*. The array *LL_port*[LL_MAXPORTS] is initialized in *pixels.h*. There is one port mask for all PLL ports: M_PLL (0xff). For more details see section B.4 (pixels.h). The

Port	***Port number***	***Port Address***
Readout Control Port	PLL_OUT0	A_PLL_OUT0
Input Port	PLL_IN0	A_PLL_IN0

following bits are defined:

DOUT_CLK	0
DOUT_LOAD	1
HITLATCHRESET	2
CCALMASKSRDIN	3

CCLKCALSR	4
CLOADCAL	5
CLOADMASK	6
NOSKIPPING	7
CAL_GATE	10
RIGHT_COL_IN	11
CSSR_CLK_IN	12
CSSR_IN	13
RESET	14
REMOTE	15
DOUT	0
DOUT_PRESENT	1

5.8 List of Functions

Unless specified otherwise, a function is defined as *int function(void)*. The action executed by a function is usually implied by the function name. *Subfunctions* are functions called by *function*. *Type* indicates whether a subfunction is a detector independent system function (*s*) or a user function (*u*) specific to either pixel or strips readout. **Coding of functions called by system functions is mandatory** (see page 59). These functions are listed in *italic*.

List of Pixel User Functions		
List of Functions Initiated by Messages		
function		
	subfunction	***type***
select_readout_mode		
	set_readout_mode(int read_mode)	*u*
load_masks		
	load_mask(int mask_mode, int mask)	*u*
reset_pac		
	writebit_LL_port(portnum, bitnum, bitval)	*s*
select_data_mode		
	fifo_data_mode = data_mode	

List of Pixel User Functions		
function		
	subfunction	***type***
get_user_code_ver		
Dispatcher		
dispatch_specific(int action_id)		
	get_verbose_flag(void) send_response(int)	*s* *s*
Setup and Initializations		
det_init		
set_defaults		
	set_sys_defaults(int *, int, int, int, int, int) set_readout_mode(int) fifo_data_mode = 0	*s* *u*
calib_init(int userpar)		
burst_det_init		
	init_user_var clear_indata_buf	*u* *u*
init_user_var		
set_readout_mode(int read_mode)		
load_mask		
	select_column(int colnum) writebit_LL_port(portnum, bitnum, bitval)	*u* *s*
select_column(int colnum)		
	writebit_LL_port(int, int, int)	*s*

List of Pixel User Functions		
function		
	subfunction	***type***
Data Collecting		
exec_calib		
	exe_trig_calib get_abort_flag(void)	*s* *s*
load_data_block(int numwords, int data[])		
	report_error(int, int)	*s*
read_det		
	read_det32 read_det52	*u* *u*
data_ready		
	read_LL_port(int, int)	*s*
Data Reporting		
report_data(int data_format)		
	assemble_header(int, int, int *) get_trig_records(int, int *, int *) export_message(int *) init_user_var	*s* *s* *s* *u*
*get_data_summary(int, int *, int *)*		
	get_data_blocks(data_mode)	*u*

5.9 Description of Pixel User Functions, Alphabetical List

int burst_det_init(void)
Executes initializations necessary before every new burst data taking cycle. Calls user function *init_user_var* and *clear_indata_buf*.

int calib_init(int userpar)
Executes user initializations. Returns 1 if *report_error*(int, int) has been called, 0 otherwise.

clear_indata_buf(void)
Clears input data buffer.

int data_ready(void)
Checks whether data is ready to be transmitted from the readout card. Returns ready-flag.

int det_init(void)
Detector initializations.

int dispatch_specific(int action_id)
Called by *execute_action*. Executes action specified by *action_id* and calls function *send_response(response_code)* if flag VERBOSE_FLAG is set to 1. Returns 1 if action executed, 0 otherwise.

int exec_calib(void)
Calibration function, called for each calibration event. Calls system function *exe_trig_calib()* to activate calibration pulse and trigger accept and returns value of flag ABORT_FLAG.

int get_data_blocks(int data_mode)
Returns number of PAC data blocks.

int get_data_summary(int trigblocks, int *flag, int *blocks)
Determines the number of data blocks **blocks*, compares it with the number of trigger blocks and sets **flag* accordingly. Calls the user function *get_data_blocks(void)*.

int get_user_code_ver(void)
This user function is called by the system main function. The function has to return the version number of the user code. Byte 0 (LSB): version of *pixels.h*, byte 1: version of *pixels.c*.

int init_user_var(void)
Initializes variables used for readout.

int load_data_block(int numwords, int data[])
Loads data in format 2 (data for link i en bloc) into memory. Returns 1 if

report_error(FATAL_ERROR,MEM_OVERFLOW) or
report_error(FATAL_ERROR, INVALID_LINK) has been called, 0 otherwise.

int load_masks(void)
Analyzes imported message, calls *load_mask(mask_mode, mask)*.

int load_mask(int mode, unsigned int bitmask[12][2])
Loads pixel bit mask. Calibration mask: $mode = 0$, disable mask: $mode \neq 0$.

int read_det(void)
Main detector readout function, called by *main* $\rightarrow$ *readout_eob* or *readout_rt*. Calls detector readout function *read_det32()* or *read_det52()* depending on the data mode selected.

int read_det32(void)
Reads 32-bits of ROC FIFO and stores 32-bit PAC data word. Repeated as long as data is available.

int read_det32(void)
Reads 52-bits of ROC FIFO and stores two 32-bit PAC data words per FIFO word. Repeated as long as data is available.

int report_data(int data_format)
Called by *data_transfer*. Assembles data in format $data\_format = 1$ or 2 and exports message(s) DATA_TRANSFER.

int reset_pac(void)
Global reset of pixel array.

int select_column(int colnum)
Selects column *colnum*.

int select_readout_mode(void)
Analyzes imported message and calls *set_readout_mode*.

int set_defaults(void)
Called by *initialize*. Sets user default values. Calls system function *set_sys_defaults* and user function *set_readout_mode* and sets data mode to 32-bit readout.

int set_readout_mode(int read_mode)
Sets readout mode.

A Instructions for Developing DSP Code on the UCI Sun

- Include the following line in your .cshrc file:
  source /u/demon/atlas/dsp/bin/dsp.cshrc

  The file *dsp.cshrc* sets environment variables for various directories and for running several scripts (see below).

- The following directories contain DSP code:

\$DSPBIN	/u/demon/atlas/dsp/bin → shell scripts and programs
\$DSPSYS	/u/demon/atlas/dsp/dspsys → DSP system code: sys*.h, sys*.c
\$DSPUCSC	/u/demon/atlas/dsp/dspstrips_ucsc → UC Santa Cruz strips code: strips.h, strips.c, stripdaq.h, stripdaq.c
\$DSPUOX	/u/demon/atlas/dsp/dspstrips_uox → University of Oxford strips code:
\$DSPLBL	/u/demon/atlas/dsp/dsppixels_lbl → LBL pixel code: pixels.h, pixels.c
\$DSPUSER	/u/demon/atlas/dsp/dspuser → User code, skeletons: user.h, user.c
\$DSPDOC	/u/demon/atlas/dsp/dspdoc postscript files of this document and sections thereof

- The following scripts compile/link code:

  - dsp_comp

    dsp_comp
    Without any argument, *dsp_comp* compiles all the *.c* files in your directory.

dsp_comp filename
dsp_comp compiles the file *filename.c*

The script redisplays the error messages if a compiling error has occured. The compiler messages are also listed in the file *ecomp_userid.err*.

- dsp_complink

  dsp_complink
  Without any argument, *dsp_complink* compiles all the *.c* files in your directory and links them to the system code.

  dsp_complink filename
  dsp_complink compiles and links file *filename.c*

  The script redisplays the compiler and/or linker error messages if any error has occured. The compiler and linker messages are also listed in the file *ecomp_userid.err*. The script controls the linking of the code by passing the command file $DSPBIN/*c30l_ie.cmd* to the linker. The output file created is named *sys_main.out*. If compilation and linking have been successful, *dsp_complink* invokes the script *getstack*. This script evaluates the number of local variables defined in the file *filename.c* or the files $*$*.c* (depending on how *dsp_complink* was invoked). It determines the total number of local variables defined (including those in the system code) and compares it with the stacksize allocated by the linker (and defined in $DSPBIN/*c30l_ie.cmd*). If the stacksize evaluated exceeds the size allocated by the linker, the script issues a warning. The script then checks whether the code fits into the memory block allocated. It then invokes the script *convertit* to convert the executable code to an ascii file to be downloaded by the VME-CPU. The name of the file created is *sys_main.asc*.

- dsp_complink_2

  dsp_complink_2 slowfile fastfile
  This script is intended for use with two user files, a file *fastfile.c* which contains time critical code (readout) and the remaining code in file *slowfile.c*. The *fastfile* will be linked into the DSP's internal RAM1. The maximum size for this code is 0x3c1.

- getstack

  getstack filename
  getstack –
  getstack filename linkcommandfile.cmd
  getstack – linkcommandfile.cmd

This script evaluates the total number of local variables defined in the system code and the file *filename.c* (*getstack filename*) or the files $*.c$ (*getstack* $-$). The second argument to the script determines whether the script compares the stacksize evaluated to the stacksize to be allocated by the linker, specified in the linker command file *linkcommandfile.cmd*. This script requires the assembly files to be present (*filename.asm*/$*.asm$). Assembly files are automatically created by the scripts *dsp_comp* and *dsp_complink*.

– convertit

convertit executable.out executable.conv
This script calls the DSP hex conversion program *hex30* to convert the executable code *executable.out* to a file *executable.conv* in ASCII Hex Format.

B Code Listings

B.1 System Code

- sys.h
- sys_glob.h
- sys_ext.h
- sys_main.c
- sys_boot.c
- sys.c
- sys_com.h
- sys_com.c
- sys_util.h
- sys_util.c

B.2 General User Code

- user.h
- user.c

B.3 Strips User Code

- strips.h
- strips.c

B.4 Pixel User Code

- pixels.h
- pixels.c

Index