The State Machine Control Package for the HERA-B Run Control System

V.Rybnikov, DESY

11 November 1999

Abstract

The State Machine Control (SMC) package described bellow is supposed to be used by all the HERA-B components for their integration into the experiment run control system.

Contents

1 Package features

The main features of the package are :

- the configuration of the process trees via configuration files;
- the possibility to have various state transition tables for different detector subsystems via loading them during the start-up;
- some automatic actions can be taken by the system in case of getting any of its subsystems in a wrong state.
- RPM/RPS interface is used for all interprocess communications.
- no user actions is necessary to deal with state transition messages exchange.

2 State control process tree

The State Control System usually has a tree-like structure.

The Central Run Control (CRC) process sends state transition commands to the component Run Control (RC) processes that transfer them to their subsystems (daughter processes). When all the subsystems are in predefined states the component RC goes to the new state and reports it to the CRC. When all the components are in the new state the CRC changes its state and the next state transition command can be sent.

The components usually contain some number of subsystems implementing different tasks: DAQ, Slow Control etc. It means that they have various number and types of internal states defined by the corresponding state transition tables (STT).

The final state of the component depends on the every subsystem state and is defined by a state dependence table (SDT). The state dependence table contains some number of expressions that determine the correspondence between the component state and the states of all its subsystems.

The following state control process classification seems to be sufficient to build-up any state control process tree:

- top process,
- leaf process,
- branch process.

2.1 Leaf process

The state of a leaf process is fully determined by its internal parameters and its state transition table. The leaf process fulfills the commands from an up-stream branch or top process.

After the start-up the leaf process goes to the initial state and reports about it to the up-stream process. The leaf process loads only its own STT.

2.2 Branch process

The state of a branch process depends on the states of its daughter branch and leaf processes (if it has them). The branch process always obeys the commands from an up-stream process. After the start-up the branch process defines its state according to the states of its daughters, goes to the corresponding state and after that reports about its state to the up-stream branch (or top) process. The branch process has to have all the STT for its daughters and SDTs of all its groups.

All the daughter processes are divided into some number of groups (usually according to their STT types). The states of the groups determine the final state of the branch process. The state dependence of each group and the branch itself are stored in SDTs filled in by the component experts.

The states of the processes belonging to one group can be considered as logical AND or logical OR, e.g. the state of the group will be effected when all the daughter processes go to the new state (AND) or at least one of them (OR). By default branch processes are in the synchronization mode. It means that if a state transition command received is not a direct state transition command (see chapter 5) then the branch process sends state transition commands "belonging" to the route one by one rather then the filnal state. Every next command is to be send only when the daughters are in the corresponding intermediate state. This mechanism solves the race conditions problem. The synchronization mode can be canceled by setting corresponding environment variable or via online command parameter usage (see discussion `smc_init`).

2.3 Top process

The main difference between a top and a branch processes is that the top process can make decisions and undertake some actions to get its branches and leafs in some states. For example the top process of the component that includes a high voltage (HV) subsystem can bring its DAQ system from RUN to PAUSE if the HV trip occurs.

Neither TOP nor BRANCH processes will be implemented on SHARCs.

3 State process stages

The state control process includes the following stages:

- start-up initialization;
- user configuration;
- working loop;
- termination.

During start-up a state control process loads the information from the following files:

- configuration file (user defined, see Appendix B and C);
- state transition tables (DAQ group defined, see Appendix D);
- state dependence tables (user defined, see Appendix E);

3.1 SMC working loop

The standard use of the SMC package is to call `smc_loop()` routine as the last one in the program that takes care of dispatching messages that don't belong to SMC_CLASS and calls user's 'main_loop' routine with the time period set by the user.

Some applications can use more than one source of information (e.g. SLP uses a SHARC link for communication over RPS and Ethernet for RPM messages). In this case the program can be built in such a way that SMC will get its messages by means arranged by the user.

The user can obtain the corresponding SMC messages handler address, or `smc_bind()` can be used for routing all SMC_CLASS messages to the corresponding handler. The user has also to use two additional SMC routines (`smc_startup(), smc_halt()`), that are usually called by `smc_loop()` (see Appendix A).

4 Component specific configuration

The following should be done by the user to customize a state machine control process:

- state transition functions should be provided and registered for corresponding state transitions. If no function has been registered for the transition than the process goes to the new state immediately;
- the handlers for all message classes used by the process should be provided and associated with the corresponding class. The user can also provide a handler for SMC_CLASS messages to change the default behavior of the package;
- user's `main_loop` routine that it to carry out the process main task has to be registered.

On Unix like platforms `main_loop` will be called every time when there is no input RPM/RPS messages and the timeout defined by the user has expired.

On Lynx OS platform `main_loop` can be started as a thread.

On SHARCs `main_loop` process is implemented as any other message handler.

In the `main_loop` routine the user can change the process state or the state error status (if something goes wrong).

5 STT and SDT formats

Every state transition table consists of two tables:

- the state definition table;
- the table of state transitions

The first table (state definition table) contains number of states and their names e.g:

12
BRAINDEAD
STOP
ERROR
INITIALIZE
...
PAUSE

The only requirements to this table are that the first state should be a meta state during the process start-up, the second - a meta state during the process termination, the third - ERROR state.

The format of the second table is the following

Number of lines		
ASCII current state	ASCII new state	ASCII temporary state

The states are given in ASCII codes.

The line that contains the temporary state as '-' determines a direct transition from the 'current state' to the 'new state'.

The line where the temporary state is a name from the state definition table determines a route to the 'new state'. These lines allows to implement a transition from one state to another one even if there is no such direct transition (see 2.2 for route transitions discussion).

All state transition tables should be stored as a global files or in a data base. At the moment SMC package considers two files name for any STT of some *type* as "*(SMCPATH)typedef.stt" forthestatedefinitiontableand*"(SMCPATH)*type*.stt" for the table of state transitions (see notes for *smc_init()*).

The BRANCH(TOP) deals with two types of state dependence table:

- SDT for the BRANCH(TOP) itself i.e. the table that defines the process state via the state of its groups;
- SDT groups for the daughter processes i.e. the table that defines the group state via the state of the daughter processes belonging to the group.

Every line of the table contains N+1 items. The first one is the process(branch,top) or group state. N is the number of groups or number of daughter processes in the group.

The daughter processes can be combined into one group if they have the same type of the state transition table. Every state dependence table has a logic parameter that points in what way the states of the processes(Groups) within the table should be considered (AND or OR).

Number of lines				
Logic parameter				
Process(Group) state	Gr(proc)state	Gr(proc)state	...	Gr(proc)state

The space of the table can be divided into some zones according to the number of states. Every zone contains some number of logical equations that define the process(Group) state via group(daughter proc) states.

The first line in the zone must contain the main condition expression for the state

$$(\text{e.g. } PAUSE = 1 \ll PAUSE\&...\&1 \ll PAUSE)$$

All the rest lines can contain any other expressions that are also true for the state

$$(\text{e.g. } PAUSE = 1 \ll PAUSE\&1 \ll RUN\&...)$$

To make the table more compact all the columns besides the first one contain the state masks e.i. not the codes of the states but one shifted to the left by the state code number. For example if code of the PAUSE state is equal 8, then the main condition expression can look like the following:

```
8 0x100 0x100 ...  0x100
```

Checking the consistency of the state transition tables and the state dependence tables is fulfilled while the process start-up.

The following criteria seem to be reasonable:

- a special 'START_UP' state must be in the state transition tables. This meta state is used to make all initializations for the process. At least one transition from the START-UP state to any other should be present in all the tables;
- a special 'EXIT' state must be in the state transition tables. This meta state is used for the process termination. At least one transition to the EXIT state should be present in all the tables;
- there should be a way out from any state besides EXIT state;
- the tables shouldn't contain states to which there is no way besides 'START_UP' state.
- all the states found in the state transition table for each process group and the process itself must be present in the state dependence table (for BRANCH and TOP processes).

6 The SMC interface routines

This section contains the descriptions of the functions that can be used by all type processes (LEAFs, BRANCHes, TOPs). Not all functions can be used by SHARC programs. The reason for that is that the usage of ASCII strings increases the size of the executable code for SHARCs drastically.

6.1 smc_add_rpm_cls()

Synopsis

```
SMC_CODES smc_add_rpm_cls(msg_type type, SMC_CODES(*fn)(void*buf,void*prm), void *prm);
```

Parameters

type	- message class (rpm_type or RPS_type), only class bits are valid.
fn()	- address of the handler that is to be called for this message class
prm	- parameter passed to the handler on every its call

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return (handler not installed)

Description

Assigns the handler for the RPM or RPS message class. Pointer to the buffer is passed to the handler as well as the parameter during the installation. Buffer contains both the header and the message itself.

A handler for SMC_CLASS messages can be assigned by the user as well. In this case the handler has to return SMC_OK if no more message processing is required. This feature cannot be used in SHARCs.

6.2 smc_add_st_fn()

Synopsis

```
SMC_CODES smc_add_st_fn(int cur_st, int new_st, SMC_CODES (*fn)(void*, void*));
```

Parameters

cur_st	- current state
new_st	- new state
fn()	- address of the function that is to be fulfilled during the state transition

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return (no such transition found)

Description

Associates the function with the state transition. The function will be called during the state transition from 'cur_st' to 'new_st'. The parameters are passed by two pointers. The first one is the pointer to the number of parameters. The second one is the pointer to the first parameter. The function has to return SMC_OK if the transition is a success, or SMC_FAILURE if the transition fails and the process should either stay in the same state or the user chooses the final process state by calling `smc_set_cur_state()` inside this routine.

The function can also return SMC_WAIT to suspend the state transition completion. In this state no any other state transition commands wiil be accepted. The user can check if there is a suspended state transition by means of `smc_if_trans_pend()`. To complete the suspended transition one has to use `smc_complete_trans()`.

The routine allows to associate the same function for different state transitions. To decrease the number of `smc_add_st_fn()` calls one can use `ANY_STATE` definition. (E.g. for invoking the same function `my_func` for all state transitions one can use the following:

```
smc_add_st_fn(ANY_STATE, ANY_STATE, my_func)
```

On Lynx and Unix like platforms the routine `smc_state_atoi()` (see bellow)is supposed to be used for retrieving 'cur_st' and 'new_st' from corresponding ASCII state names.

6.3 smc_bind()

Synopsis

```
SMC_CODES smc_bind(void);
```

Return values

SMC_OK - on completion

Note

The routine is not to be used in *SHARCs*.

Description

Binds the SMC message handler for LEAF or TOP/BRANCH processes to all messages of SMC_CLASS. The function is supposed to be used with `rpm_dispatch()`.

6.4 smc_complete_trans()

Synopsis

```
SMC_CODES smc_complete_trans(void);
```

Return values

SMC_OK	- state transition completed
SMC_ERROR	- error detected

Description

Tries to complete the pending transition if it exists via calling the corresponding state transition function (see. `smc_add_st_fn()`.

6.5 smc_errstat_itoa()

Synopsis

```
char* smc_errstat_itoa(int st);
```

Parameters

st - status code

Return values

char* - pointer to the ASCII string

Note

The routine is not to be used in *SHARCs*

Description

The routine allows to obtain the ASCII error status name if the code status is defined for SMC. `INVALID` is returned if the code is not defined.
The current definition of the error statuses is the following:

- `NORMAL 0x0`
- `WARNING 0x1`
- `ALARM 0x2`
- `ERROR 0x4`
- `UNKNOWN 0x8`

6.6 smc_get_cmd_prm()

Synopsis

```
SMC_CODES smc_get_cmd_prm(int cur_st,int new_st,int**prm,int*prmn);
```

Parameters

cur_st	- current state
new_st	- new state
prm	- pointer to get the address of the first parameter
prmn	- pointer to get the number of parameters

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return (no such transition found)

Description

Allows to obtain the parameters for the transition from 'cur_st' to 'new_st'. In case the parameters are unknown the number of parameters returned is -1.

6.7 smc_get_cur_state()

Synopsis

```
int smc_get_cur_state(void);
```

Return values

int	- the current state code

Description

Allows to obtain the current state of the process.

6.8 smc_get_err_status()

Synopsis

```
int smc_get_err_status(void);
```

Return values

int	- the error status code of the current state

Description

Allows to obtain the error status of the process current state.

6.9 smc_get_new_state()

Synopsis

```
int smc_get_new_state(void);
```

Return values

int	- the new state code

Description

Allows to obtain the new state of the process.

6.10 smc_get_reg_name()

Synopsis

```
char* smc_get_reg_name(void);
```

Return values

char*	- address of the ASCII string containing the process name registered by RPM.

Note

The routine is not to be used in *SHARCs*

Description

Allows to obtain the process name registered by RPM.

6.11 smc_get_sts_prm()

Synopsis

```
void smc_get_sts_prm(int**prm,int*prmn);
```

Parameters

prm	- pointer to get the address of the first parameter
prmn	- pointer to get the number of parameters

Return values

NONE

Description

Allows to obtain the parameters for the last state transition.

6.12 smc_halt()

Synopsis

```
void smc_halt(void);
```

Description

Should be called before program termination if the program uses smc_startup() instead of smc_loop().

6.13 smc_handler_addr()

Synopsis

```
SMC_CODES (*smc_handler_addr(void))(void*, void*);
```

Note

The routine is not to be used in *SHARCs*

Description

The routine returns the address of the SMC message handler for the LEAF or TOP/BRANCH processes. The handler has two input parameters:

- pointer to the buffer containing the header and the body of the message;
- pointer to a parameter (currently not used);

The handler always returns `SMC_OK`.

The routine has to be used by the programs that don't use `smc_loop()`.

6.14 smc_handler_b_addr()

Synopsis

```
SMC_CODES (*smc_handler_b_addr(void))(rmp_hdr*, void*, void*);
```

Note

The routine is not to be used in *SHARCs*

Description

The routine returns the address of the SMC message handler for the LEAF or TOP?BRANCH processes. The routine differes from `smc_handler_addr()` by the first and second parameters that are to be passed to the handler.
Tey are:

- pointer to the buffer containing the header of the message;
- pointer to the buffer containing the body of the message;
- pointer to a parameter (currently not used);

The handler always returns `SMC_OK`.

The routine has to be used by the programs that don't use `smc_loop()`.

The routine is to be used in case of `rpm_bind()` usage.

6.15 smc_if_trans_pend()

Synopsis

```
SMC_CODES smc_if_trans_pend(void);
```

Return values

SMC_OK	- there is a pending transition
SMC_ERROR	- no pending transition

Description

Checks if there is a pending transition (see. `smc_add_st_fn()`).

6.16 smc_init()

Synopsis

```
SMC_CODES smc_init(int argc, const char** argv);
```

Parameters

argc	- argument count from command line
argv	- command line parameters

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return
SMC_NOFILE	- no file found (STT or SDT)
SMC_BAD_DATA	- bad data format in one of the input files

Description

Initializes the SMC package. This function should be called before any other call to SMC routines. RPM or RPS interfaces should be initialized beforehand.

The function can use either a configuration file or some switches and process environment variables.

The following switches and environment variables are defined for the function:

- *For all processes*

-SMCCFG:name	SMCCFG	- process configuration file name;
-SMCPTP:type	SMCPTP	- process type (TOP, BRANCH, LEAF);
-SMCDBG:level	SMCDBG	- debug level (0, 1, 2);
-SMCSTT:type	SMCSTT	- process STT (number);
-SMCRCS:#	SMCRCS	- number of RPM classes (number);
-SMCPCI:type	SMCPCI	- process command interface (RPM,KEYBOARD);
-SMCCPN:#	SMCCPN	- number of command processes (number);
-SMCPNM:name	SMCPNM	- the command process RPM name;
-SMCLOG	SMCLOG	- to set error logging flag;
-SMCPATH:path	SMCPATH	- to set path for STT and SDTs
-SMCPRCHST	SMCPRCHST	- to take into account the privilege (must be SMC_PRIV_ALL) of the up-stream process for the commands SMC_STATECHG and SMC_EXIT

- *For TOPs and BRANCHEs*

-SMCSDT:name	SMCSDT	- process SDT file (name);
-SMCGCLS#:name	SMCGCLS#	- RC class of the group numbered # (name);
-SMCGNB:#	SMCGNB	- number of process groups (number);
-SMCGSDT#:name	SMCGSDT#	- SDT file name of the group numbered # (name);
-SMCGSTT#:type	SMCGSTT#	- STT of the group numbered # (number);
-SMCGDNB#:num	SMCGDNB#	- number of processes in the group numbered # (number);
-SMCDNM#:name	SMCDNM#	- daughter process RPM name included in the group

		numbered #.
-SMCNOSYNCH	SMCNOSYNCH	- cancels the synchronization mode
-SMCDIST	SMCDIST	- switch on the state transition commands distribution mode
-SMCTMT	SMCTMT	- state transition time-out, if state transition is not complete, a new state transition command will be sent
-SMCPRCFG	SMCPRCFG	- to take into account the privilege (must be SMC_PRIV_ALL) of the up-stream process for the command SMC_CONFIG

If a configuration file switch is used (-SMCCFG:) then the configuration file is to be read. According to the information from the file the state transition table of the process and its daughters are loaded. For the TOP and BRANCH processes the state dependence tables are loaded for the process and all daughter groups as well . The consistencies of all STT and SDT are checked. The debug level and logging can be set as well.

If one doesn't use the configuration file then some switches or environment variables are supposed to be used. In general case the environment variables supersede the corresponding switches. The table bellow contains the information concerning the switches and environment variables.

Switch(env)	Usage	Default value
-SMCPTP:	optional	LEAF
-SMCDBG:	optional	0
-SMCSTT:	optional	7
-SMCRCS:	optional	5
-SMCPCI:	optional	RPM
-SMCCPN:	optional	4
-SMCPNM:	optional	see notes bellow
-SMCLOG	optional	Off
-SMCPATH:	optional	see notes bellow
-SMCSDT:	optional	gr1_7.sdt
-SMCGNB:	optional	1
-SMCGCLS#:	optional	-
-SMCGSDT#:	optional	gr2_7.sdt
-SMCGSTT#:	optional	7
-SMCGDNB#:	obligatory	-
-SMCDNM#:	obligatory	see notes bellow
-SMCNOSYNCH	optional	On
-SMCDIST	optional	Off
-SMCTMT	optional	300 sec
-SMCPRCHST	optional	Off
-SMCPRCFG	optional	Off

Notes

If the name of the command process is not given or the number of command processes (-SMCCPN:) is more then given names (-SMCPNM:) then one more name is assigned to the process - 'xxxx_monitor'. Where xxxx is RMP name of the process.

The number of the daughters process names (-SMCGDNB#:) can be more than those set by (-SMCDNM#:, where # is the group number). In this case all the rest processes are marked as 'NOT ACTIVE' and will not be considered by SMC.

The default SMCPATH is set during the compilation and usually points to the */etc/* subdirectory of the online library root directory. (i.e. for AFS machines "/afs/dsy/group/herab/online/xxx/$(BINTYPE)/etc/", xxx = dev or pro). Switch *-SMCPATH:* supersedes the appropriate environment variable.

If the switch -SMCDIST is used then the state transition commands will be distributed to the daughter processes one by one when the prevoius ones are already in the new state. This can reduce the network traffic in case the daughter processes are running on many machines connected to one cluster and the network is being used heavily during the state transitions.

6.17 smc_loop()

Synopsis

For SHARCs:

```
void smc_loop(int argc, const char **argv);
```

For the rest world:

```
void smc_loop(SMC_CODES(*m_loop)(), struct timeval time, int argc, const char ** argv);
```

Parameters

argc	- the argument count from command line, if available
argv	- command line parameters, if available
m_loop(*void)	- the address of the main loop routine defined by the user that is called during idle loops or started as a thread (Lynx OS)
time	- sets the period of calling 'm_loop' if it is not a thread

Description

The routine should be called as the last one in the program. The initialization transition is carried out and the process goes to the first state according to its STT.

For SHARCs the process main job is fulfilled by handlers registered through `smc_add_rpm_cls()` calls.

For all the rest world, the user defined routine is called with the set time period or this routine is to be started as a thread.

6.18 smc_print_all_stt()

Synopsis

```
void smc_print_all_stt(void);
```

Note

The routine is not to be used in *SHARCs*

Description

Prints all state transition tables known to the process. For TOP and BRANCH processes the tables for their daughter processes will be printed out as well.

6.19 smc_print_proc_prm()

Synopsis

```
void smc_print_proc_prm(void);
```

Note

The routine is not to be used in *SHARCs*

Description

Prints all the process parameters according to the configuration file, switches or environment variables.

6.20 smc_send_info()

Synopsis

```
void smc_send_info(void* buffer, int len, int id);
```

Parameters

buffer	- the address of the message
len	- the message length
id	- the message id (used by RPS)

Note

The routine is not to be used in *SHARCs*

Description

The routine allows to send a message to the all up-stream processes by means of SMC. The format of the message has to be registered over `smc_set_info_format()` beforehand. The message type is SMC_INFO.

On getting SMC_INFO message type the SMC will pass it to the `m_loop()` (see `smc_loop()`) routine as `info_msg` structure. One more possibility to obtain this message is to install SMC_CLASS message handler (see `smc_add_rpm_cls()`).

6.21 smc_set_cur_state()

Synopsis

```
SMC_CODES smc_set_cur_state(int new_st);
```

Parameters

new_st	- a new state for the process

Return values

SMC_OK	- normal successful return
SMC_ERROR	- the state transition is not allowed (no such transition in the STT)
SMC_WAIT	- there is a pending state transition

Description

The routine allows to set a new state or stop the state transition from a user routines during state transitions or while 'main_loop'.

6.22 smc_set_cur_new_state()

Synopsis

```
SMC_CODES smc_set_cur_new_state(int cur_st, int new_st);
```

Parameters

cur_st	- a current state for the process
new_st	- a new state for the process

Return values

SMC_OK	- normal successful return
SMC_ERROR	- one of the states (or both) not found in STT)
SMC_FAILURE	- the process is not a LEAF process

Description

The routine allows to set the current and new states. No matter if the state transition is allowed and if the process was in transition before. No state transition function is called.

6.23 smc_set_err_status()

Synopsis

```
void smc_set_err_status(int errstat);
```

Parameters

errstat - error status of the state for the process

Description

The routine allows to set the error status of the current state. The new error status will propagate up to the top of the process tree.

6.24 smc_set_info_format()

Synopsis

```
SMC_CODES smc_set_info_format(char *format);
```

Parameters

format	- format of the message (see `rpm_add_def()`)

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return

Note

The routine is not to be used in *SHARCs*

Description

The routine assigns 'format' for SMC_INFO message type.

6.25 smc_start_thread()

Synopsis

```
void smc_start_thread(void);
```

Note

The routine is to be used in *Lynx* applications.

Description

Sets the request for starting user's `m_loop()` (see `smc_loop`) or `func()` (see `smc_startup()`) as a thread.

The routine has to be called after `smc_init()`.

6.26 smc_startup()

Synopsis

For SHARCs:

```
void smc_startup(int argc, const char ** argv);
```

For the rest world:

```
void smc_startup(SMC_CODES(*func)(), int argc, const char ** argv);
```

Parameters

argc	- the argument count from command line, if available
argv	- command line parameters, if available
func(*void)	- the address of the function defined by the user that is to be called or started as a thread (Lynx OS)

Description

If the user doesn't use `smc_loop()` then `smc_startup()` has to be used as the last one from the point of view SMC package. The routine fulfills the state transition from the START-UP state according to the STT, checks the up-stream processes and sends them the process state. The routine calls user defined function that can also be started as a thread (Lynx) if the corresponding request has been set (see `smc_start_thread()`).

6.27 smc_state_itoa()

Synopsis

```
char* smc_state_itoa(int st);
```

Parameters

st	- state code

Return values

char*	- pointer to the ASCII string

Note

The routine is not to be used in *SHARCs*

Description

The routine allows to obtain the ASCII state name if the code state is found in the process STT. 'UNKNOWN' is returned if the code is not found.

6.28 smc_state_atoi()

Synopsis

```
int smc_state_atoi(char* st_name);
```

Parameters

st_name - ASCII string of the state name

Return values

int - code of the state

Note

The routine is not to be used in *SHARCs*

Description

The routine converts the given state name to the state code. '-1' is returned if the state name is not found. This function gives the possibility not to deal with the coded states at all for Lynx and Unix like operating systems (see `smc_add_st_fn()`).

7 The SMC interface routines for branch and top processes

7.1 smc_add_proc_in_group()

Synopsis

```
SMC_CODES smc_add_proc_in_group(char* name,int grn,int act);
```

Parameters

name	- the daughter process name
grn	- the group number
act	- the activity mask

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return

Description

Allows to add a process with the name 'name' to the group with number 'grn'. Process can be added if there is an empty entry in the process group table. In case of success the activity mask of the added process is to be set to 'act'.

7.2 smc_get_cp_num()

Synopsis

```
int smc_get_cp_num(void);
```

Return values

int	- maximum number of command processes

Description

Returns the maximum number of command processes

7.3 smc_get_cpt()

Synopsis

```
SMC_CPT* smc_get_cpt(void);
```

Return values

SMC_CPT* - address of the first command process table

Description

Allows to obtain the address of the first command process table. This allows to obtain information about all command processes outside SMC.

7.4 smc_get_dproc_name()

Synopsis

```
char* smc_get_dproc_name(rpm_hdr* hdr);
```

Parameters

hdr	- the header address of the received message

Return values

char*	- the daughter process name address

Description

Allows to retrieve the daughter process name corresponding to the source address contained in the received message header.

7.5 smc_get_group_num()

Synopsis

```
int smc_get_group_num(void);
```

Return values

int	- number of groups

Description

Returns the number of process groups

7.6 smc_get_pgt()

Synopsis

```
SMC_PGT* smc_get_pgt(void);
```

Return values

SMC_PGT* - address of the first process group table

Description

Allows to obtain the address of the first process group table.

7.7 smc_get_pst()

Synopsis

```
SMC_PST* smc_get_pst(void);
```

Return values

SMC_PST* - address of the process state table

Description

Allows to obtain the address of the process state table.

7.8 smc_print_all_sdt()

Synopsis

```
void smc_print_all_sdt(void);
```

Description

Prints all state dependence tables known to the process

7.9 smc_print_daughter_state_list()

Synopsis

```
void smc_print_daughter_state_list(SMC_DSTL* dlist);
```

Parameters

dlist - pointer to the daughter states list structure

Description

Prints the contents of the daughter states list.

7.10 smc_rem_proc_from_group()

Synopsis

```
SMC_CODES smc_rem_proc_from_group(char* name,int grn);
```

Parameters

name	- the daughter process name
grn	- the group number

Return values

SMC_OK	- normal successful return
SMC_ERROR	- error return

Description

Allows to remove a process with the name 'name' from the group with number 'grn'. The process name is set to NULL in the process group table, the activity mask - to 'PROC_REM'.

The state of the group is checked up. If the state of the group has been changed then the process state is to be checked. If the process state has been changed then the new state is to be sent to all up-stream processes.

7.11 smc_send_ping_all()

Synopsis

```
void smc_send_ping_all(void);
```

Description

Send the message of type SMC_STATEALLRQ to all daugther processes.

7.12 smc_send_ping_in_trans()

Synopsis

```
void smc_send_ping_in_trans(void);
```

Description

Send the message of type SMC_STATEALLRQ to all daugther processes that are in transition
.

7.13 smc_send_ping_to_one()

Synopsis

```
void smc_send_ping_to_one(int group, int proc);
```

Parameters

group	- process group number
proc	- process number in the group

Return values

- SMC_FAILURE	- process is a LEAF
- SMC_NOMORE	- group or proc is invalid
- SMC_NOPROC	- process has invalid RPM addr
- GRR_...	- any GRR erro code

Description

Send the message of type SMC_STATEALLRQ to a particular process.

7.14 smc_set_cp_num()

Synopsis

```
void smc_set_cp_num(int ncp);
```

Parameters

ncp - number of command processes

Description

Sets the number of command processes (see description for `smc_set_cpt()`.)

7.15 smc_set_cpt()

Synopsis

```
void smc_set_cpt(SMC_CPT* cpt);
```

Parameters

cpt - the address of the command process table.

Description

Sets the address of the first command process table. This routine together with `smc_set_cp_num()` provide the possibility to create the command process tables outside the SMC and pass the table parameters to it (the number of the processes and the address of the first table). This mechanism allows to configure SMC top and branch processes without configuration files information concerning command processes.

7.16 smc_set_group_num()

Synopsis

```
void smc_set_group_num(int ngr);
```

Parameters

ngr - number of groups

Description

Sets the number of process groups (see description for `smc_set_pgt()`.)

7.17 smc_set_pgt()

Synopsis

```
void smc_set_pgt(SMC_PGT* pgt);
```

Parameters

pgt - the address of the first process group table.

Description

Sets the address of the first process group table. This routine together with `smc_set_group_num()` provide the possibility to create the process group tables outside the SMC and pass the table parameters to it (the number of the groups and the address of the first table). This mechanism allows to configure SMC top and branch processes without configuration files information concerning daughter processes.

8 Appendix A. Example of a component program based on SMC package

The examples bellow (`prog1.c`, `prog1a.c`, `prog2.c`) can be used for leaf DAQ processes. The current DAQ state transition table (see Appendix B) is used in the example.

The example prog1.c runs on Linux, IRIX and Lynx OS platforms. In case of Lynx OS the main user loop `my_clock()` can be started as a thread. In case of Linux it is called every 10 seconds when the program is idle.

The differences between `prog1.c` and `prog1a.c` are the following.

- `smc_startup()` and `smc_halt()` are used in `prog1a.c` instead of `smc_loop()`;
- to be able to pass the SMC messages to the SMC handler `smc_handler_addr()` is called;
- for Lynx OS the request to start a thread `func()` is used.
- the user takes care of getting all events in the `while()` loop.
- the usage of `smc_bind()` and `rpm_dispatch()` is shown as commentaries.

The example `prog2.c` runs on SHARCs.

Both examples are pure state control machine processes that don't deal with the other RPM or RPS class messages. They can be easily added via `smc_add_rpm_cls()` calls.

All the state transition routines are gathered in `sttrfunc.c` and the prototypes are in `sttrfunc.h`. The only state transition function for the transition $BRAINDEAD \rightarrow INITIALIZE$ is registered in the main programs. All the rest ones are registered in the above mentioned function. All the RPM or RPS message handlers can be registered as well in that function or in the main programs.

Three state transition functions are different from all the rest.

The function `safe_to_init()` fails every second time. Therefore the process will not go to the INITIALIZE state when the failure occurs.

The function `run_to_ready()` fails every third time in such a way that process goes to the ERROR state.

The function `pause_to_run()` prints out the input parameters.

For Lynx OS and Unix like platforms one can use the state names instead of state codes. The lines like

```
 if(smc_add_st_fn(0,3, dead_to_init)) {
  return;
}
```

can be changed to

```
 if(smc_add_st_fn(smc_state_atoi("BRAINDEAD"),
                  smc_state_atoi("INITIALIZE"),
                  dead_to_init)) {
  return;
}
```

```
 ------------------------- prog1.c -----------------------------------
/*
****+
**  FACILITY:  HERA-B_STATE_MACHINE_CONTROL_PACKAGE
**
**  MODULE DESCRIPTION:
**
**      State machine control package test program.
**
**  AUTHORS:
**
**      V.Rybnikov
```

```
**
**  CREATION DATE: 08/09/1997
**
**  MODIFICATION HISTORY:
**
**  08/09/97    VR  first version
**  09/10/97    VR  Threads for Lynx added
**  15/10/97    VR  All state transition functions are moved to
'sttrfunc.c'
**--
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#ifdef Lynx
#include <ipc.h>
#include <pthread.h>
#endif
#include "smc.h"
#include "sttrfunc.h"

SMC_CODES my_clock(void*);

void main(int argc, const char **argv) {
int status;
struct timeval tm;

  status = rpm_init(argc, argv);
  if(status != GRR_OK) {
    if(d_level != SMC_NO) {
      printf("rpm_init - failed ");
    }
    return;
  }

  status = rpm_open(RPM_REGISTER);
  if(status != GRR_OK) {
    if(d_level != SMC_NO) {
      printf("rpm_open - failed ");
    }
    return;
  }

  status = smc_init ( argc, argv);
  if(status != SMC_OK) {
    printf("smc_init failed ");
    return;
  }

  if(smc_add_st_fn(0,3, dead_to_init)) {
    return;
  }
  tm.tv_sec = 10;
  tm.tv_usec = 0;
```

```
  smc_loop(my_clock, &tm, argc, argv);

  rpm_halt();
}


/*****************************************************************
**   Obtaining current time
**
**   Returns: SMC_OK
**
*****************************************************************/

SMC_CODES my_clock(void *arg) {
static int count = 0;
time_t t;
int thread = 0;
char my_time[32];

 if((int*)arg) thread = 1;

 do{
   count++;
   t = time(NULL);
   memset(my_time, 0, sizeof(my_time));
   strcpy(my_time, ctime(&t));
   my_time[strlen(my_time)-1] = 0;
   printf("clock: %s, cur.st:%s new.st:%s", my_time ,
          smc_state_itoa(smc_get_cur_state()),
          smc_state_itoa(smc_get_new_state()));
   fflush(0);
   if(thread) sleep(5);
#ifdef Lynx
   pthread_yield();
#endif
 }while(thread);

 return SMC_OK;
}
 ------------------------- prog1a.c -----------------------------------
/*
**++
**  FACILITY:  HERA-B_STATE_MACHINE_CONTROL_PACKAGE
**
**  MODULE DESCRIPTION:
**
**      State machine control package test program.
**
**  AUTHORS:
**
**      V.Rybnikov
**
**  CREATION DATE: 08/09/1997
**
**  MODIFICATION HISTORY:
**
**  08/09/97    VR  first version
**  09/10/97    VR  Threads for Lynx added
**  15/10/97    VR  All state transition functions are moved to 'sttrfunc.c'
```

```
**  25/02/98    VR  For those who doesn't want to use smc_loop()
**--
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#ifdef Lynx
#include <ipc.h>
#include <pthread.h>
#endif
#include "smc.h"
#include "sttrfunc.h"

SMC_CODES my_clock(void*);

void main(int argc, const char **argv) {
int status;
struct timeval tm;

char msg[RPM_LMAX + sizeof(rpm_hdr)];
rpm_hdr *hdr = (rpm_hdr*)msg;
SMC_CODES (*smc_handler)(void*, void*);

  status = rpm_init(argc, argv);
  if(status != GRR_OK) {
    if(d_level != SMC_NO) {
      printf("rpm_init - failed ");
    }
    return;
  }

  status = rpm_open(RPM_REGISTER);
  if(status != GRR_OK) {
    if(d_level != SMC_NO) {
      printf("rpm_open - failed ");
    }
    return;
  }

  status = smc_init ( argc, argv);
  if(status != SMC_OK) {
    printf("smc_init failed ");
    return;
  }

#ifdef Lynx
  smc_start_thread();
#endif

  if(smc_add_st_fn(0,3, dead_to_init)) {
    return;
  }

  status =  smc_startup(
#ifdef Lynx
```

```
func,
#else
NULL,
#endif
 argc, argv);

  if(status != SMC_OK){
    printf("smc_startup failed\n");
    return;
  }

  smc_handler = smc_handler_addr();

/* or simply -> smc_bind(); */

  tm.tv_sec = 10;
  tm.tv_usec = 0;

  rpm_ctl(RPM_RECEIVE_TIME,(void*)&tm.tv_sec);
  while(smc_get_cur_state() != EXIT_STATE) {

 /*    user specific code
          ......
          ......

 */
    status = rpm_receive(hdr, (void*)&msg[sizeof(rpm_hdr)], RPM_LMAX,
                         BLOCK_TIL_TIMEOUT);
    if(status == GRR_TIME) {
 /*    user specific code
          ??????
          ??????
 */
    } else if(status == GRR_OK) {
      if(((hdr->type) >>16) == SMC_CLASS) {
        (* smc_handler)((void*)msg,(void*)0);
      } else {
        printf("RPM_CLASS: %X\n", hdr->type);

 /*    user specific code
          ||||||
          ||||||
 */
      }

/* or symply -> rmp_dispatch(hdr, (void*)(msg + sizeof(rpm_hdr)));*/

    } else {
 /*    user specific code
          !!!!!!
          !!!!!!
 */
    }

  }

  smc_halt();
  rpm_halt();
}
```

```
void func() {
 /*    user specific code
          \\\\\\
          //////
 */
}


 ------------------------- prog2.c --------------------------------
/*
**++
**  FACILITY:  HERA-B_STATE_MACHINE_CONTROL_PACKAGE
**
**  MODULE DESCRIPTION:
**
**      State machine control package test program.
**
**  AUTHORS:
**
**      V.Rybnikov
**
**  CREATION DATE: 08/09/1997
**
**  MODIFICATION HISTORY:
**
**  08/09/97    VR  first version
**  21/10/97    VR  SHARCs specific features added
**--
*/

#include "sio.h"
#include "smc.h"
#include "sttrfunc.h"

void main(int argc, const char **argv) {
  int rc;

  if((rc=RPS_register(argv[0], RPS_my_address())) != RPS_ok) {
    printf("RPS register failed with %d! ",rc);
    return;
  }
  if((rc=smc_init (argc, argv) )!= SMC_OK){
    printf("smc init failed with %d! ",rc);
    return;
  }
  if((rc=smc_add_st_fn(0,3, dead_to_init))) {
    printf("smc_add_st_fn failed with %d! ",rc);
    return;
  }

  smc_loop(argc, argv);
  RPS_remove(argv[0]);
}
 ------------------------- sttrfunc.h ----------------------------

#ifndef __sttr_h__
#define __sttr_h__
#include "smc.h"
```

```
extern SMC_CODES dead_to_init(void*, void*);

extern SMC_CODES init_to_config(void*, void*);
extern SMC_CODES init_to_stop(void*, void*);

extern SMC_CODES config_to_safe(void*, void*);
extern SMC_CODES config_to_init(void*, void*);
extern SMC_CODES config_to_stop(void*, void*);

extern SMC_CODES safe_to_config(void*, void*);
extern SMC_CODES safe_to_init(void*, void*);
extern SMC_CODES safe_to_prepare(void*, void*);
extern SMC_CODES safe_to_stop(void*, void*);

extern SMC_CODES prepare_to_safe(void*, void*);
extern SMC_CODES prepare_to_ready(void*, void*);

extern SMC_CODES ready_to_prepare(void*, void*);
extern SMC_CODES ready_to_begin(void*, void*);

extern SMC_CODES begin_to_run(void*, void*);

extern SMC_CODES run_to_pause(void*, void*);
extern SMC_CODES run_to_ready(void*, void*);

extern SMC_CODES pause_to_ready(void*, void*);
extern SMC_CODES pause_to_run(void*, void*);

#endif

 ------------------------- sttrfunc.c ----------------------------
#ifdef _my_sharc_
#include "sio.h"
#else
#include <stdio.h>
#endif
#include "smc.h"
#include "sttrfunc.h"

SMC_CODES dead_to_init(void*a, void*b){

  if(smc_add_st_fn(3, 4, init_to_config)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(3, 1, init_to_stop)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(4,5, config_to_safe)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(4,3, config_to_init)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(4,1, config_to_stop)) {
    return SMC_ERROR;
  }
```

```
  if(smc_add_st_fn(5,4, safe_to_config)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(5,3, safe_to_init)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(5,6, safe_to_prepare)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(5,1, safe_to_stop)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(6,5, prepare_to_safe)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(6,7, prepare_to_ready)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(7,6, ready_to_prepare)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(7,10, ready_to_begin)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(10,9, begin_to_run)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(9,8, run_to_pause)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(9,7, run_to_ready)) {
    return SMC_ERROR;
  }

  if(smc_add_st_fn(8,7, pause_to_ready)) {
    return SMC_ERROR;
  }
  if(smc_add_st_fn(8,9, pause_to_run)) {
    return SMC_ERROR;
  }


  printf("dead_to_init: DONE ");
  return SMC_OK;
}

SMC_CODES init_to_config(void*a, void*b){

  printf("init_to_config: DONE ");
  return SMC_OK;
}
SMC_CODES init_to_stop(void*a, void*b){

  printf("init_to_stop: DONE ");
  return SMC_OK;
```

```
}

SMC_CODES config_to_safe(void*a, void*b){

  printf("config_to_safe: DONE ");
  return SMC_OK;
}

SMC_CODES config_to_init(void*a, void*b){

  printf("config_to_init: DONE ");
  return SMC_OK;
}

SMC_CODES config_to_stop(void*a, void*b){

  printf("config_to_stop: DONE ");
  return SMC_OK;
}


SMC_CODES safe_to_config(void*a, void*b){

  printf("safe_to_config: DONE ");
  return SMC_OK;
}

SMC_CODES safe_to_init(void*a, void*b){
static int step = 0;
  if(step) {
    step = 1;
    printf("safe_to_init: FAILED ");
    return SMC_FAILURE;
  }
  step++;
  printf("safe_to_init: DONE ");
  return SMC_OK;
}

SMC_CODES safe_to_prepare(void*a, void*b){

  printf("safe_to_prepare: DONE ");
  return SMC_OK;
}

SMC_CODES safe_to_stop(void*a, void*b){

  printf("safe_to_stop: DONE ");
  return SMC_OK;
}


SMC_CODES prepare_to_safe(void*a, void*b){

  printf("prepare_to_safe: DONE ");
  return SMC_OK;
}

SMC_CODES prepare_to_ready(void*a, void*b){
```

```
  printf("prepare_to_ready: DONE ");
  return SMC_OK;
}


SMC_CODES ready_to_prepare(void*a, void*b){

  printf("ready_to_prepare: DONE ");
  return SMC_OK;
}

SMC_CODES ready_to_begin(void*a, void*b){

  printf("ready_to_begin: DONE ");
  return SMC_OK;
}


SMC_CODES begin_to_run(void*a, void*b){

  printf("begin_to_run: Ok ");

  return SMC_OK;
}


SMC_CODES run_to_pause(void*a, void*b){

  printf("run_to_pause: DONE ");
  return SMC_OK;
}

SMC_CODES run_to_ready(void*a, void*b){
static int step = 0;

  if(step == 2) {
    step = 0;
    smc_set_cur_state(5);
/***************************************************************/
/*          !!!     For all platforms but SHARCs       !!!      */
/* you can write smc_set_cur_state(smc_state_atoi("ERROR"));    */
/***************************************************************/
    printf("run_to_ready: FAILED ");
    return SMC_FAILURE;
  }
  step++;
  printf("run_to_ready: DONE ");
  return SMC_OK;
}


SMC_CODES pause_to_ready(void*a, void*b){

  printf("pause_to_ready: DONE ");
  return SMC_OK;
}

SMC_CODES pause_to_run(void*a, void*b){
```

```
int i;
int *d = (int*)b;

  printf("pause_to_run: input parameters: ");
  for(i=0; i < *(int*)a; i++)
    printf("%d ", d[i]);
  printf("pause_to_run: DONE ");
  return SMC_OK;
}
```

9 Appendix B. Examples of configuration files for leaf processes

Two examples of the configuration files for leaf processes are given bellow.

The first can be used for leaf processes in test mode when one wants to have a quick test over the keyboard control.

The state transition table *type* (12) will be translated by SMC to the file names "./*12*def.stt" and "./*12*.stt".

The leaf can contain up to 3 RPM message handlers.

The second example can be used for leaf processes that are controlled over the RPM interface and has more than one up-stream process. In the example the up-stream processes are supposed to be registered under names 'sct' or 'smc_monitor'.

For SHARCs the configuration file is a C code. `leaf2.cfg.c` is the code that configures the process in the same way as `leaf2.cfg` for of Linux or Lynx OS. The only difference is the number of RPS message handlers that is limited my the RPS package.

```
------------------------ leaf1.cfg -----------------------------------
# Start SMC configuration #
# Process type #
LEAF
# State Transition Table Type #
12
# Number of RPM classes #
3
# Command input interface #
KEYBOARD
#####END#####
------------------------ leaf2.cfg -----------------------------------
# Start SMC configuration #
# Process type #
LEAF
# State Transition Table Type #
12
# Number of RPM classes #
3
# Command input interface #
RPM
# Number of processes #
2
# Names of processes #
/sct
/smc_monitor
#####END#####
------------------------ leaf2.cfg.c ---------------------------------
int NUM_PROC = 2;
char *PROC_NAMES[]= {
"sct",
"smc_monitor"
};
```

10 Appendix C. Example of a configuration file for top and branch processes

This branch process deals with two processes with the same state transition table *type* (12).

The process state dependence table file is `gr1_1.sdt`. The group SDT file is `gr1.sdt`.

All the state dependence files are given in Appendix E

```
-------------------------- branch.cfg --------------------------------
# Start SMC configuration #
# Process type #
BRANCH
# State Transition Table Type #
12
# State dependence table file name #
gr1_1.sdt
# Number of RPM classes #
3
# Daughters #
# Number of groups #
1
# Group_1 #
# RC class #
BRANCH_FLT
# State Transition Table Type #
12
# State dependence table file name #
gr1.sdt
# Number of processes #
2
# Names of processes #
/sct11
/sct12
# Command input interface #
RPM
# Number of processes #
1
# Names of processes #
/smc_monitor
#####END#####
```

11 Appendix D. Example of a state transition table

The contents of the file `12.stt` is given bellow. It contains current DAQ state transition table. The state definitions are given in `12def.stt`.

For SHARCs leaf processes the corresponding C code (`12.stt.c`) should be used.

Besides direct transitions (upper part of the tables) the tables contain two routes:

$$INITIALIZE \rightarrow RUN,\ READY \rightarrow STOP.$$

```
------------------------------ 12def.stt ---------------------------
11
BRAINDEAD
STOP
ERROR
INITIALIZE
CONFIGURE
SAFE
PREPARE
READY
PAUSE
RUN
BEGIN
------------------------------ 12.stt ------------------------------
33
BRAINDEAD INITIALIZE -
INITIALIZE STOP -
INITIALIZE CONFIGURE -
CONFIGURE STOP -
CONFIGURE INITIALIZE -
CONFIGURE SAFE -
SAFE STOP -
SAFE INITIALIZE -
SAFE CONFIGURE -
SAFE PREPARE -
ERROR SAFE -
PREPARE SAFE -
PREPARE READY -
READY PREPARE -
READY BEGIN -
PAUSE READY -
PAUSE RUN -
RUN READY -
RUN PAUSE -
BEGIN RUN -
SAFE ERROR -
PREPARE ERROR -
READY ERROR -
PAUSE ERROR -
RUN ERROR -
BEGIN ERROR -
INITIALIZE RUN CONFIGURE
CONFIGURE RUN SAFE
SAFE RUN PREPARE
PREPARE RUN READY
READY RUN BEGIN
READY STOP PREPARE
PREPARE STOP SAFE
```

```
------------------------------ 12.stt.c ----------------------------
#include "sio.h"
#include "smc.h"
int TBL_LEN = 33;
int TBL_TYPE = 12;
SMC_STL TBL[] = {
{0, 3, -1, NULL},
{3, 1, -1, NULL},
{3, 4, -1, NULL},
{4, 1, -1, NULL},
{4, 3, -1, NULL},
{4, 5, -1, NULL},
{5, 1, -1, NULL},
{5, 3, -1, NULL},
{5, 4, -1, NULL},
{5, 6, -1, NULL},
{2, 5, -1, NULL},
{6, 5, -1, NULL},
{6, 7, -1, NULL},
{7, 6, -1, NULL},
{7, 10, -1, NULL},
{8, 7, -1, NULL},
{8, 9, -1, NULL},
{9, 7, -1, NULL},
{9, 8, -1, NULL},
{10, 9, -1, NULL},
{5, 2, -1, NULL},
{6, 2, -1, NULL},
{7, 2, -1, NULL},
{8, 2, -1, NULL},
{9, 2, -1, NULL},
{10, 2, -1, NULL},
{3, 9, 4, NULL},
{4, 9, 5, NULL},
{5, 9, 6, NULL},
{6, 9, 7, NULL},
{7, 9, 10, NULL},
{7, 1, 6, NULL},
{6, 1, 5, NULL}
};
```

12 Appendix E. Examples of state dependence tables

The file `gr2_12.sdt` gives the rules for defining the group state process depending on the state of two processes.

The file `gr1_12.sdt` sets the rules for defining the BRANCH (TOP) process state depending on the state of one group.

```
------------------------------ gr2_12.sdt ---------------------------
30
AND
0 0x1 0x7ff  BRAINDEAD
0 0x7ff 0x1  BRAINDEAD
1 0x2 0x2  STOP
1 0x2 0x7fe  STOP
1 0x7fe 0x2  STOP
2 0x4 0x4  ERROR
2 0x4 0x7fc  ERROR
2 0x7fc 0x4  ERROR
3 0x8 0x8  INITIALIZE
3 0x8 0x7f8  INITIALIZE
3 0x7f8 0x8  INITIALIZE
4 0x10 0x10  CONFIGURE
4 0x10 0x7f0  CONFIGURE
4 0x7f0 0x10  CONFIGURE
5 0x20 0x20  SAFE
5 0x20 0x7e0  SAFE
5 0x7e0 0x20  SAFE
6 0x40 0x40  PREPARE
6 0x40 0x7c0  PREPARE
6 0x7c0 0x40  PREPARE
7 0x80 0x80  READY
7 0x80 0x780  READY
7 0x780 0x80  READY
8 0x100 0x100  PAUSE
8 0x100 0x300  PAUSE
8 0x300 0x100  PAUSE
9 0x200 0x200  RUN
10 0x400 0x400  BEGIN
10 0x400 0x700  BEGIN
10 0x700 0x400  BEGIN
------------------------------ gr1_12.sdt ---------------------------
11
AND
0 0x1  BRAINDEAD
1 0x2  STOP
2 0x4  ERROR
3 0x8  INITIALIZE
4 0x10  CONFIGURE
5 0x20  SAFE
6 0x40  PREPARE
7 0x80  READY
8 0x100  PAUSE
9 0x200  RUN
10 0x400  BEGIN
```

13 Appendix F. Headers and libraries

The headers and libraries are available on the following clusters:

```
HERA-B: header - /hb/daq/Online_Library/Rev0005/include/smc.h;
        library - /hb/daq/Online_Library/Rev0005/lib/librcs.a
```

```
hb-con: header - /home/hb/Rev0005/include/smc.h;
        library - /home/hb/Rev0005/lib/librcs.a
```

```
hb-vme: header - /home/herab/Rev0005/include/smc.h;
        library - /home/herab/Rev0005/lib/librcs.a
```

```
the up to date versions can be found:
        directory - /afs/desy.de/user/r/rybnikov/public/smc/
        source    - smc.c
        header    - smc.h
        library   - smc.o
```

For Unix like platforms in oder to link SMC functions from `librcs.a` the library `libgrr.a` and `libmsg.a` are also need.

To compile the SMC source for SHARCs programs the symbol `_my_sharc_` should be defined.

14 Appendix G. Tools for monitoring, control and development

The TclTk based program `smcctl` located on **HERA-B** in `/hb/daq/Online_Library/Rev0005/bin/` and on **hb-con** in `/home/hb/Rev0005/bin` can be used both for monitoring and control of the SMC based program. The program has a built-in help. The up to date version of the `smcctl` one can find on **HERA-B** in `/afs/desy.de/user/r/rybnikov/public/smc`.

The TclTk based program `bsdt` located on **HERA-B** in `/afs/desy.de/user/r/rybnikov/public/smc` can be used for ctreating SDTs. The only requirements to work with the program is that the STTs, that one is going to use, should be stored in the working directory. The programm allows to extent SDTs with two processes in a group to any number of processes in a group.

The program `smc_toshfm` located on **HERA-B** in `/afs/desy.de/user/r/rybnikov/public/smc` converts the configuration files and STTs from the text formats (see Appendixes B,D) to C code used by leaf processes running on SHARCs.

15 Appendix H. Switches usage examples

```
> sct
```

Comments:

- All parameters are set to the defaults (see `smc_init()`)

- Command process name `sct_monitor` (set internally, see discussion on `smc_init()`)

```
> sct -STT:8 -SMCPNM:my_monitor
```

The same as the first example but:

- STT of *type* 8 will be loaded,
- process with RPM name 'my_monitor' will be also included as the command process.

```
> sct -SMCPTP:BRANCH -SMCSDT:mypr.sdt -SMCDBG:2 -STT:8 \
       -SMCPNM:my\_monitor -SMCGSDT1:mygr.sdt -SMCGDNB1:3 \
       -SMCDNM1:dtr1 -SMCDNM1:dtr2
```

The same as the second example but:

- process type is BRANCH,
- process SDT - `mypr.sdt`,
- number of groups is default (1),
- the group SDT - `mygr.sdt`,
- number of processes in the first group is 3, their names: dtr1, dtr2. The third process has no name and will be masked as 'NOT ACTIVE'.