



API for TRACE32 Instruction Set Simulator

[TRACE32 Online Help](#)

[TRACE32 Directory](#)

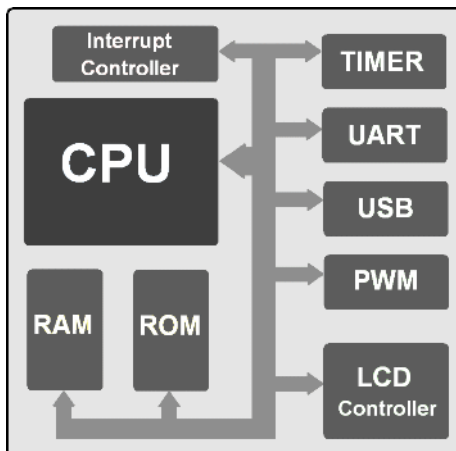
[TRACE32 Index](#)

TRACE32 Documents	
TRACE32 Instruction Set Simulators	
API for TRACE32 Instruction Set Simulator	1
Overview	2
Peripheral Simulation Model	6
Standard function	6
Registers	10
Timers	13
Stall	14
Ports	15
Terminals	17
Communication	18
Files	18
Deprecated functions	20
Practical script commands	21
Peripheral model example	25
Environment	25
Source code listing	28
Initialization scripts	37

This document describes the implementation process and the basic use of peripheral simulation model in TRACE32.

Overview

PSM - Peripheral Simulation Model contains functions and registers of corresponding physical modules supported by MCU. It is additional program (overlay) for simulated core. To complete the tasks, core operates on registers located in physical memory of MCU. PSM is a software overlay for memory area occupied by peripheral module. For an accurate simulation of microprocessor's unit, interaction between a core and peripherals module is required. PSM provides functions responsible for interaction between core and other modules. PSM is able to simulate any module which significantly increases functionality of the entire simulation environment.



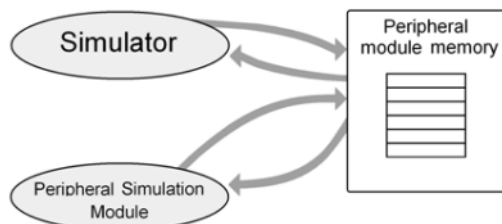
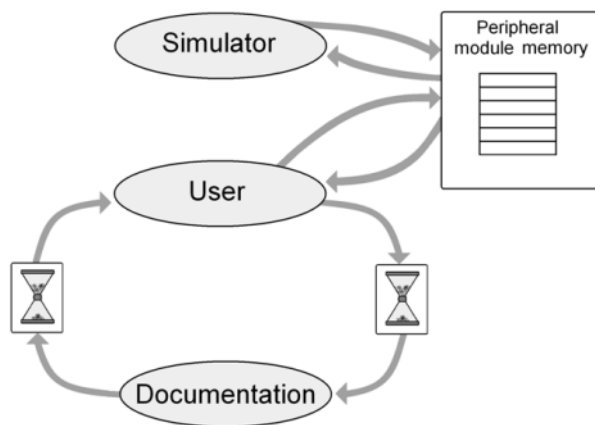
The processor retrieves and sends data from / to memory shared with other MCU modules, hence the need to simulate not only the processor core, but also cooperative modules.

Simulation of microprocessor's system without the module requires manual setting of the appropriate bits in registers of peripheral module (important for proper software operation). For proper registers' configuration to which a processor sends / receives data, user have to perform time-consuming work with a documentation. Bits in a registers are frequently modified with an each clock cycle.

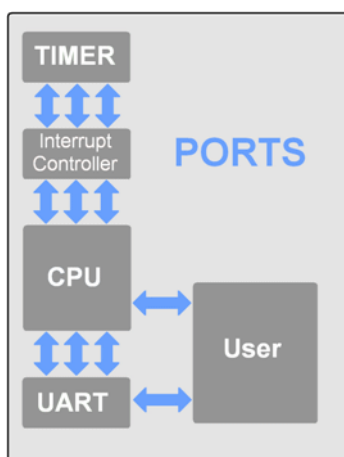
By Peripheral Simulation Models complicated settings are done automatically. Software takes over the role of an interaction between model and simulator and between other modules. Using PSM increases possibility of checking a software functionality in implemented microprocessor's system. An important feature of this solution is to eliminate a **human error factor** in process of modifying registers contents while working with documentation.

Simulation **without** Peripheral Simulation Models.

Simulation **with** Peripheral Simulation Models.

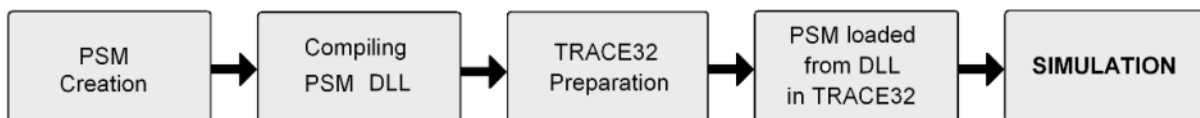


Ports are used for communication between models, processor simulator and user. Currently there are over 512 ports. Ports are the primary communication interface provided by TRACE32.



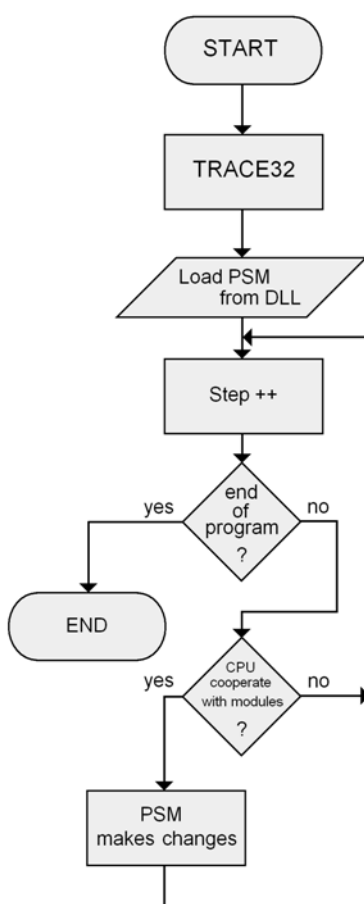
PSM is stored in a DLL. The functions and resources of the DLL can be used directly by TRACE32. The DLL is not an independent program. Libraries are dynamically imported into the memory at the time specified by a programmer (mostly when actually needed), hence a definition of dynamically linked library. DLL files are often used in programs as plug-ins.

The module contained in a DLL can be loaded using the appropriate command in TRACE32

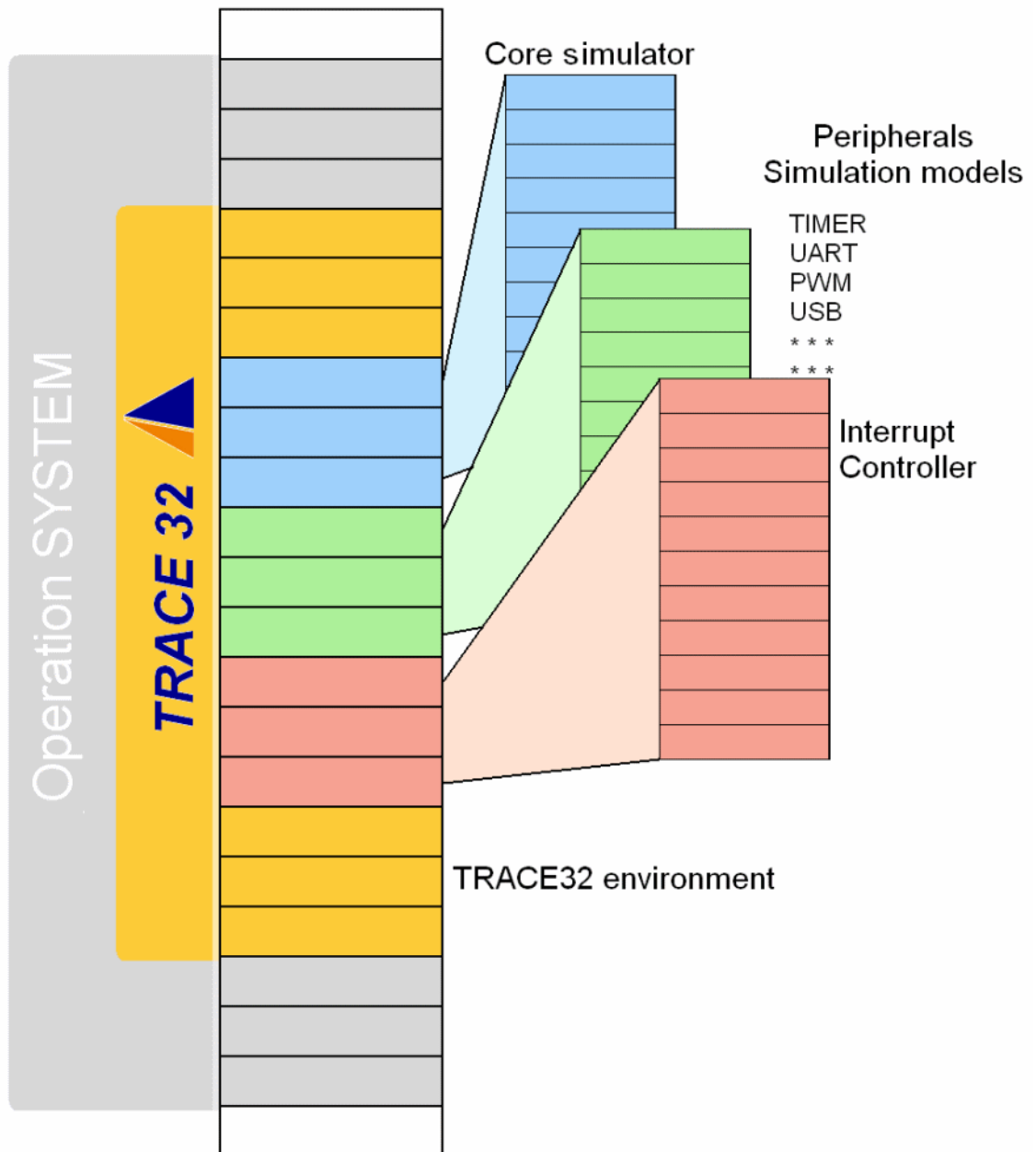


Simulation process:

Microprocessor's system is often simulated by executing sequence of steps and observing the program's response. In simulated system are some operations CPU performs itself, such as arithmetic-logic operations, copying, etc. Those operations do not need to cooperate with other modules and program may be continued. Nevertheless when it comes to cooperation between CPU and other modules (most cases) it is necessary to ensure adequate interaction in a simulator. PSM performs its functions (data change in registers, interruption send, decrementation, etc.) and after those functions are completed, program is continued until the very end.



For better understanding of simulation operating rules it is advised to remember that the simulation is a virtual representation of real processes and takes place in computer memory. TRACE32 simulation environment allows to simulate any system. TRACE32 reserves area of memory for a simulated system and in this area simulator performs operations. All performed operations and created variables are stored in TRACE32 memory.



Peripheral Simulation Model

This section describes: functions frequently used in the model, operations on the registers, communication in the simulation, IO operations and the basic commands in a initialization script.

Standard function

Callback functions are used to handle events such as reset, exit from TRACE32, GO,BREAK, timers, changes on ports lines and changes of the registers' contents in model memory (see the file simul.h). These functions are called when registered events are performed, so it is required to initialize them in a **SIMUL_Init** function. The memory allocation for our variables should be done inside **SIMUL_Init** function (do not use global variables). Allocation is done with **SIMUL_Alloc** function.

Event handler functions:

1. Event reset callback function.

```
SIMUL_RegisterResetCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle simulation model reset.

2. Event read callback function.

```
SIMUL_RegisterBusReadCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private, int bustype, simulWord *  
addressFrom, simulWord * addressTo);
```

Register callback function to handle register events in simulation model. Function responsible for reading register (separately for each of registers).

3. Event write callback function.

```
SIMUL_RegisterBusWriteCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private, int bustype, simulWord *  
addressFrom, simulWord * addressTo);
```

Register callback function to handle register events in simulation model. Function responsible for writing to register (separately for each of registers).

4. Event port change callback function.

```
SIMUL_RegisterPortChangeCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private, int offset, int width);
```

Register callback function to handle port changes in simulation model.

5. Event command callback function.

```
SIMUL_RegisterCommandCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle a commands in TRACE32 environment. This function returns SIMUL_COMMAND_OK or SIMUL_COMMAND_FAIL.

6. Event exit from TRACE32 callback function.

```
SIMULAPI SIMUL_RegisterExitCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle exit from a model in TRACE32 environment. This function returns SIMUL_EXIT_OK.

7. Event GO callback function.

```
SIMULAPI SIMUL_RegisterGoCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle GO command in TRACE32 environment. This function returns SIMUL_GO_OK.

8. Event Terminal callback function.

```
SIMUL_RegisterTerminalCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private, int id);
```

Register callback function to handle terminal in TRACE32 environment.

9. Event break callback function.

```
SIMULAPI SIMUL_RegisterBreakCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle BREAK command in TRACE32 environment. This function returns SIMUL_BREAK_OK.

10. Reallocation (changes) register address function.

```
SIMUL_RelocateBusCallback(simulProcessor processor, void * callbackid, int  
bustype, simulWord * paddressFrom, simulWord * paddressTo);
```

Register callback function to handle addresses changes in simulation model registers in TRACE32 environment.

11. Register timer function.

```
SIMUL_RegisterTimerCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private);
```

Register callback function to handle timers in a simulation model in TRACE32 environment.

Explanation of terms appearing in the functions:

func	Function that supports an event
private	Global variable of our model
bustype	Variable of the structure of CBS
offset	Port from which the start
width	Amount of ports
paddressFrom	Start address of register
paddressTo	End address (paddressFrom + regsize)

Please note that the maximum number of ports (**width**) for a single registration is 32, to declare other ports **offset** variable have to be changed.

The model can be loaded with parameters (types and kinds of parameters can be defined by the user), e.g. base module address, number of interrupts in INTC etc.

Access to this data can be obtained through the structure **simulCallbackStruct*cbs** in an initialization function.

cbs-> x.init.argc	Indicated the number of parameters (value “1” means no parameters)
cbs-> x.init.argpbustype[1]	Indicates the type of bus. Value of that variable should stored in the main structure as a bustype variable (for future use)
cbs-> x.init.argpport [x]	stores parameters of int type
cbs-> x.init.argp [x]	stores parameters of char* type

The exact description of structure is in ‘simul.h’ file.

There are standard functions which are designed to handle the model. For example, display of warning messages, memory allocation, etc. Standard functions:

1. Formated displaying.

```
SIMUL_Printf(simulProcessor processor, const char *format, ...);
```

Function used to display text message in TRACE32 environment.

2. Displaying warnings in TRACE32 command line.

```
SIMUL_Warning(simulProcessor processor, const char *format, ...);
```

Function used to display warnings in TRACE32 environment.

3. Stopping simulation in TRACE32.

```
SIMUL_Stop(simulProcessor processor);
```

Function used to stop simulation in TRACE32 environment.

4. Updated data TRACE32.

```
SIMUL_Update(simulProcessor processor, int flags);
```

Function used to update data in TRACE32 environment.

5. Allocation of memory for variables.

```
SIMUL_Alloc(simulProcessor processor, int size);
```

Function used to allocate memory in TRACE32, for variables used in a simulation model.

6. Deallocate memory for variables.

```
SIMUL_Free(simulProcessor processor, void * ptr);
```

Function used to free memory in TRACE32 environment.

7. Get current endian (Little = 0).

```
SIMUL_GetEndianness(simulProcessor processor);
```

Function used to get a current endian settings used in a simulation model.

Registers

Model should simulate all types of registers:

RO	read only register
R/W	read write register
W	write register
RC	read clear by 0 register
R1C	read clear by 1 register
R/W	read write register

For each register, there should be two functions (depending on register access type). One responsible for reading and one for writing a certain area of memory (register). These functions should be in **Simul_Init** function in 'model_name'.c file (these are callback functions).

For each function that supports register (memory) designer can determine what kind of access type is being executed by TRACE32, whether it is being read by peripheral file or being accessed by CPU. According to values in the variable **cbs->x.bus.cycletype**:

SIMUL_MEMORY_HIDDEN	Access peripheral file or simulation bus
SIMUL_MEMORY_DATA	CPU access through instructions such as load
SIMUL_MEMORY_FETCH	CPU load opcode
SIMUL_MEMORY_DMA	DMA

Register type 'RO' is simulated by not executing callback function responsible for writing. The register type 'W' is simulated by not executing callback function responsible for reading, while other registers are simulated in similarly way according to privileges.

Functions operating on registers in model:

1. Read memory function.

```
SIMUL_ReadMemory(simulProcessor processor, int bustype, simulWord *  
paddress, int width, int cycletype, simulWord * pdata);
```

Function used to read memory from specified area in TRACE32 environment.

2. Write memory function.

```
SIMUL_WriteMemory(simulProcessor processor, int bustype, simulWord *  
paddress, int width, int cycletype, simulWord * pdata);
```

Function used to write memory to specified area in TRACE32 environment.

3. Insert function (smaller word to a greater word). Such as 8-bit to 32-bit.

```
SIMUL_InsertWord(simulProcessor processor, simulWord * ptarget, int  
wordwidth, simulWord * paddress, int width, simulWord * pdata);
```

Function used to insert a smaller word in the greater word.

4. Extract function (smaller word from a greater word). Such as 8-bit from 32-bit.

```
SIMUL_ExtractWord(simulProcessor processor, simulWord * psource, int  
wordwidth, simulWord * paddress, int width, simulWord * pdata);
```

Function used to extract a smaller width word from a greater width word.

5. Data save function.

```
SIMUL_SaveWord(simulProcessor processor, void * ptarget, int width,
simulWord * pdata);
```

Function used to save a value of specified word.

6. Data load function.

```
SIMUL_LoadWord(simulProcessor processor, void * psource, int width,
simulWord * pdata);
```

Function used to load a value from specified word.

Explanation of terms appearing in the functions:

bustype	type of bus
paddress	address
width	width in bits
wordwidth	width of the object on which operations are executed
cycletype	access type, eg hidden
pdata	data to write / read
ptarget	target to save
psource	source to load

In supported registers functions it is required to set a **cbs-> x.bus.clocks** variable which is responsible for number of cycles of access to these functions.

The **cbs.x.bus->address** variable stores address of read or write. The access type is stored in **cbs.x.bus>width** variable - it is possible to write an 8, 16-bit or 32-bit register. To avoid misunderstandings it is recommended to write a functions to handle hazardous situations, or use **SIMUL_InsertWord / SIMUL_ExtractWord** when these registers are supported. Write / read data are transmitted in **cbs.x.bus>data** variable.

At the end function returns **SIMUL_MEMORY_OK**.

Timers are used to measure time and are calculated relative to TRACE32. Each timer used in a model have to be registered in the initialization function (Simul_Init). Timer needs to be started using **SIMUL_StartTimer** function and can be stopped using **SIMUL_StopTimer** function. It is also possible to retrieve a current simulation time etc.

Functions operating on timers:

1. Start timer function. Function used to start selected timer in the simulation model.

```
SIMUL_StartTimer(simulProcessor processor, void * timerid, int mode,
simulTime * ptime);
```

2. Stop timer function. Function used to stop selected timer in the simulation model.

```
SIMUL_StopTimer(simulProcessor processor, void * timerid);
```

3. Stop all timers function. Function used to stop all timers in the simulation model..

```
SIMUL_StopAllTimer(simulProcessor processor);
```

4. Read the time since the start of the simulation.

```
SIMUL_GetTime(simulProcessor processor, simulTime * ptime);
```

5. Read number of clock cycles since the start of the simulation.

```
SIMUL_GetClock(simulProcessor processor, int clockid, simulTime * pclock);
```

6. Read the frequency of the specified clock.

```
SIMUL_GetClockFrequency(simulProcessor processor, int clockid, simulWord64
* pfrequency);
```

7. Set clock frequency function

```
SIMUL_SetClockFrequency(simulProcessor processor, int clockid, simulWord64
* pfrequency);
```

8. Reads the time for one clock cycle.

```
SIMUL_GetClockCycle(simulProcessor processor, int clockid, simulTime *
ptime);
```

9. Sets the time for one clock cycle.

```
SIMUL_SetClockCycle(simulProcessor processor, int clockid, simTime *  
ptime);
```

Explanation of terms appearing in the functions:

timerid	timer identifier
mode	timer mode
ptime	pointer to time variable (in picoseconds)
pclock	pointer to variable to receive clock number
func	function responsible for timer operation
clockid	clock identifier
pfrequency	pointer to frequency variable (in Hz / cycle per second)

Timer mode depending on '**mode**' variable:

SIMUL_TIMER_ABS	ptime specifies absolute time calculated from the beginning of simulation
SIMUL_TIMER_REL	ptime specifies the number of clock cycles from now
SIMUL_TIMER_SINGLE	single call
SIMUL_TIMER_REPEAT	call at specific timer
SIMUL_TIMER_CLOCKS	clock is increased with each step of simulator

For example, if a mode is set on (**SIMUL_TIMER_REPEAT** | **SIMUL_TIMER_CLOCKS**) then simulator calls a timer with each step (depends on how many cycles equal one step on simulator such as ARM = 3).

Stall

The stall function can be used to stall the core (prevent instruction fetch/execution) for a certain time.

```
SIMUL_Stall(simulProcessor processor, int mode, simTime * ptime);
```

Explanation of terms appearing in the functions:

mode	timer mode
ptime	defines set / returns value

The meaning of '**ptime**' depends on '**mode**' variable:

SIMUL_STALL_ABS	ptime specifies absolute time calculated from the beginning of simulation
SIMUL_STALL_REL	ptime specifies the number of clock cycles from now
SIMUL_STALL_CLOCKS	clock is increased with each step of simulator

Ports

The ports serve as communication between different models and also between models and a simulator.

To set port value in TRACE32:

port.set p.0 high	sets value '1' on port 0.
port.set p.0 low	sets value '0' on port 0.

Currently, there are 512 general-purpose ports and 8 (-1 to -8) special ports dedicated for communication with simulator. To use ports in model designer has to register callback functions in Simul_Init in 'model_name'.c file.

Number of ports used in specific function is defined by setting width variable. By using two variables from cbs structure (**cbs-> x.port.newdata** and **cbs-> x.port.olddata**) we can find out whether there is change and on which port. For more than one port, use function to determine which port has changed its value.

Functions operating on ports:

1. Set value port function.

```
SIMUL_SetPort(simulProcessor processor, int offset, int width, simulWord *  
pdata);
```

2. Get value port function.

```
SIMUL_GetPort(simulProcessor processor, int offset, int width, simulWord *  
pdata);
```

3. Port change function.

```
SIMUL_RegisterPortChangeCallback(simulProcessor processor,  
simulCallbackFunctionPtr func, simulPtr private, int offset, int width);
```

Explanation of terms appearing in the functions:

offset	port number
width	number of ports
pdata	set /get value
func	function that supports an event
private	variable output model

At the end function returns **SIMUL_PORT_OK**.

Terminal is used to display data, thus it is possible to simulate the graphic driver or similar device displaying information in a dedicated window.

Functions operating on terminal:

1. Current state function.

```
SIMUL_StateTerminal(simulProcessor processor, int id);
```

2. Read data function.

```
SIMUL_ReadTerminal(simulProcessor processor, int id);
```

3. Write characters function.

```
SIMUL_WriteTerminal(simulProcessor processor, int id, int ch);
```

Explanation of terms appearing in the functions:

id	terminal ID
private	model variable
ch	character to write on terminal

Terminal state is returned by following value.

SIMUL_STATE_RXREADY	Terminal ready to read
SIMUL_STATE_TXREADY	Terminal ready to write
SIMUL_STATE_NOTEXISTING	There is no terminal

Basic interface used for communication consists of general-purpose ports (0 to 511), dedicated ports lines - 1 (reset), -2 (interruption), -3 (NMI) [exact names can be found in Ports section in simul.h file] and registered callback functions.

Model should report an interrupt event to simulator using a dedicated port. For interrupt handling in a simulator (only for Interrupt Controller), please write appropriate functions. (if does not exist). Interrupt processing should be confirmed by clearing a dedicated line. All accesses to memory (peripherals, registers) have to be made through access type **SIMUL_MEMORY_DATA**, unless they are special sequence of read / write, implemented in the target system through bus. In that case **SIMUL_MEMORY_HIDDEN** access type should be used. Functions operating on normal type of access is already written in a simulator. However for special type of access new function have to be written, bearing in mind here that there are no variables associated with CTS.

Files

In PSM it is possible to operate on files, without using functions provided by standard header files. It is important because TRACE32 acts on many platforms and operating systems. Therefore, functions defined in simul.h file are used to let TRACE32 support various platforms and systems.

Functions operating on files:

1. Open file function.

```
SIMUL_OpenFile(simulProcessor processor, const char * filename, int mode);
```

2. Close file function.

```
SIMUL_CloseFile(simulProcessor processor, void * file);
```

3. Read file function.

```
SIMUL_ReadFile(simulProcessor processor, void * file, void * pdata,  
int length);
```

4. Write file function.

```
SIMUL_WriteFile(simulProcessor processor, void * file, void * pdata,  
int length);
```

5. Read line from file function.

```
SIMUL_ReadlineFile(simulProcessor processor, void * file, void * pdata,  
int length);
```

6. Write line to file function.

```
SIMUL_WritelineFile(simulProcessor processor, void * file, void * pdata);
```

7. Set file position (returns position after operation).

```
SIMUL_SeekFile(simulProcessor processor, void * file, long pos, int mode);
```

Explanation of terms appearing in the functions:

filename	Name of file
file	File ID
mode	Access mode
length	Length of write / read data
pdata	Data pointer
pos	Position to set

Mode variable has following values for **OpenFile**:

SIMUL_FILE_READ	Read a file
SIMUL_FILE_WRITE	Write a file

SIMUL_FILE_CREATE	Create a file)
SIMUL_FILE_APPEND	Adding to a file
SIMUL_FILE_BINARY	Binary file
SIMUL_FILE_ASCII	ASCII file

Mode variable has following values for **SeekFile**:

SIMUL_FILE_SEEKABS	Setting position in relation to the beginning of file
SIMUL_FILE_SEEKREL	Setting position in relation to the current position in file
SIMUL_FILE_SEEKEND	Setting position in relation to the end of file

Deprecated functions

Below functions are deprecated and shall not be used in new implementations. The functions are kept as part of the interface to maintain backwards compatibility.

```
SIMUL_CreateSharedResource(simulProcessor processor, int size);
```

```
SIMUL_ReadSharedResource(simulProcessor processor, int offset, int len,
void * pdata);
```

```
SIMUL_WriteSharedResource(simulProcessor processor, int offset, int len,
void * pdata);
```

Practical script commands

To properly support the model in TRACE32 and easier usage model by other people, initscript should be created. Task of initialization scripts is prepare model to work with a simulator as if the initialization has been performed by MCU procedure in TRACE32.

Initscripts are created in files with the extension *.CMM.

Commands used in initialization script for Peripheral simulation model (more details see [General Commands Reference Guide](#)):

1. Start command and End command.

DO	Start script
ENDDO	End of script

Command "DO" begin the script, and command "ENDDO" inform of the completion.

2. CPU select command.

```
Format:          SYSTEM.CPU <cpu>

<cpu>:           C64X, ARM7TDMI, ARM9E, C6455,...
```

Specify the exact CPU type used on your target. This is required to get the matching PER file and other CPU specific settings (e.g. predefined settings for on-chip FLASH).

3. Port name set command.

```
Format:          NAME.SET port.<number> <port_name>

<number>:       -8,-7,...511,512.

<port_name>:    interrupt, timer, model,...
```

Gives name of selected port. All ports used in system have to be listed.

4. Deletes the specified model from TRACE32 memory command.

```
Format:          SIM.UNLOAD <path to model_name.dll>

<path to        arm_timer.dll, vic_lpc.dll,...
model_name
.dll>:
```

Selected simulation model is removed from TRACE32 memory. Command without specified model, clears all simulation models from TRACE32 memory.

5. Loads the specified model to TRACE32 memory command.

```
Format:          SIM.LOAD <model_name.dll> [parameters]

<path           ../model/arm_timer.dll, c:/vic_lpc.dll,...
dll>:

[parameter      "timer" "cpu"
s]:
```

Selected simulation model is loaded into TRACE32 memory. If the model has parameters, then have to be provided. In other action model can be unpredictable.

6. System up command.

```
Format:          SYSTEM.UP
```

After this command all libraries are initialized.

7. Performs an operation on the address command.

```
Format:          DATA.ASSEMBLE [<address>] <mnemonic>

<address>:      0xff000000, 0xbc80a000,...

<mnemonic>      nop, subs, str, move,...
:
```

Data.Assemble is used to replace the code at memory <address> with the assembler instruction specified by <mnemonic>; <mnemonic> describes the instruction with respect to the CPU-mnemonic.

8. Displaying instruction window command.

Format: **DATA.LIST**

Display format (in assembler, mixed or HLL) is selected dynamically, depending on the current debug mode. If no address is specified, the window tracks to the value of the program counter. The window is only scrolled, if the bar moves outside of a predefined subwindow.

9. Displays memory area in a window command.

Format: **DATA.DUMP** <range>

<range>: **0x0--0xff, 0xffff8000--0xffff90a0, ...**

If a single address is selected this address will define the windows' initial position only. Scrolling makes other memory contents visible. When selecting an address range only the defined data range can be dumped. Range definition is useful whenever the addresses following are read protected (e.g., in the case of I/O) or more than one page will be printed. This command allows to preview the memory area. The area is updated every step. When a window is displayed reading is made with each step.

10. Sets a value in the specified register command.

Format: **REGISTER.SET** <register> <value>

<
register>: **r0, r1, pc, ...**

<value>: **0x4, 20, 1, 0xf, 0, ...**

This command sets a given value into the selected register. The register names differ for each processor architecture.

11. Set a breakpoint at address command.

Format: **BREAK.SET** <address>

<address>: **0xff000000, 0xbc80a000, ...**

Command useful for observing system behavior in certain program spots. Without parameters the command opens a dialog window for setting breakpoints.

12. Sets a value with a specified width, at the selected address command.

```
Format:          DATA.SET <address> <width> <value>

<address>:       0xff000000, 0xbc80a000, ...

<width>:         %byte,%word,%long, ...

<value>:         0x4,20,1,0xf,0, ...
```

This command allows to set the memory area value. For example, registers are reset from selected area simultaneously or overwritten any other value.

13. Opens the specified PER file command.

```
Format:          PER <path_to_per_file>

<path_to_per_file>: ../per/c64.per, C:/arm/per/ARM9.per, ...
```

Command opens PER file under the specified location. The peripherals of integrated microcontrollers can be displayed and manipulated with the command **PER**. The command offers a free configurable window for displaying memory or i/o structures. So it is possible to display the state of peripheral chips or memory based structures very comfortably.

14. Introduce program code into the simulator command.

```
Format:          DATA.PROGRAM <address> <path_to_asm_file>

<address>:       0xff000000, 0xbc80a000, ...

<path_to_per_file>: ../program.asm, C:/code/program.asm, ...
```

This command creates a window for editing and assembling a short assembler program. Without a specified filename the file T32.ASM is generated. If the Compile button is used, syntax errors and undefined labels will be detected. The resulting program will be assembled for the specified address and saved to memory. The labels entered will be added to the symbol list.

Peripheral model example

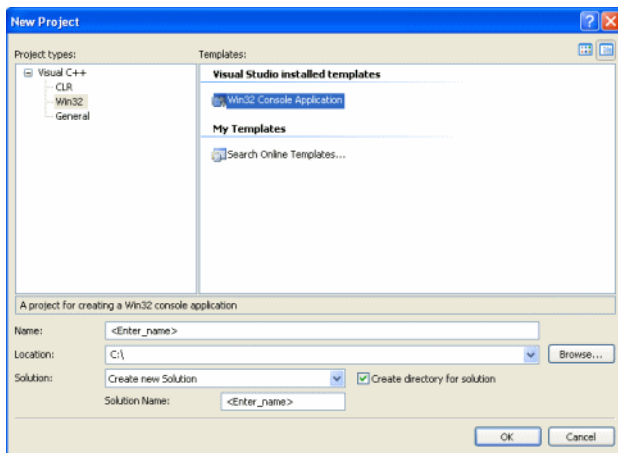
Practical example of timer simulating model is designed to show the whole process of programming, implementation and maintenance.

Environment

Follow these steps:

In Visual Studio create 'New Project' by choosing File -> New -> Project...

In a "New Project" window, select Win32 -> Win32 Console Application

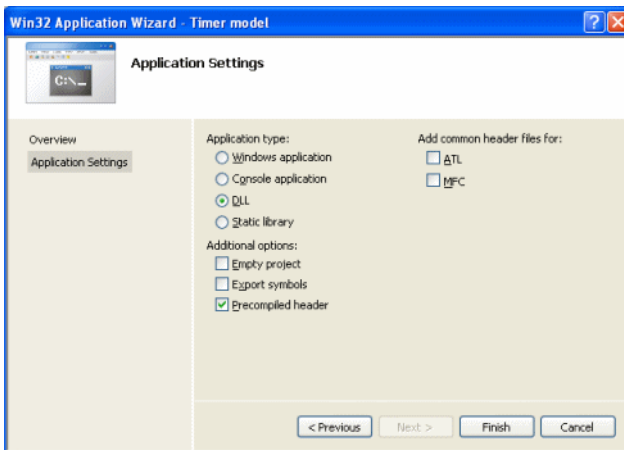
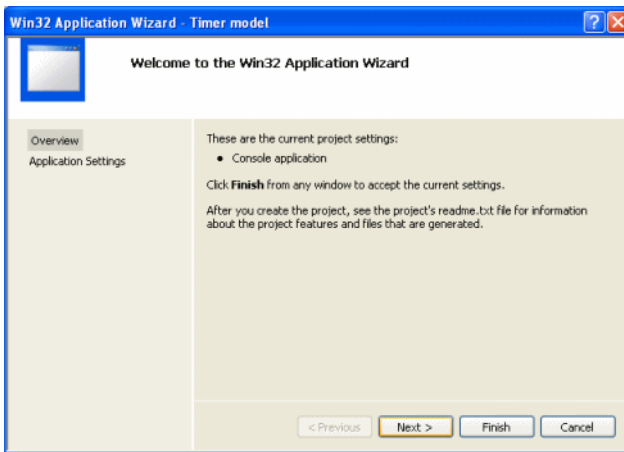


In 'Name' field, enter the name of new model (in this case is a "timer").

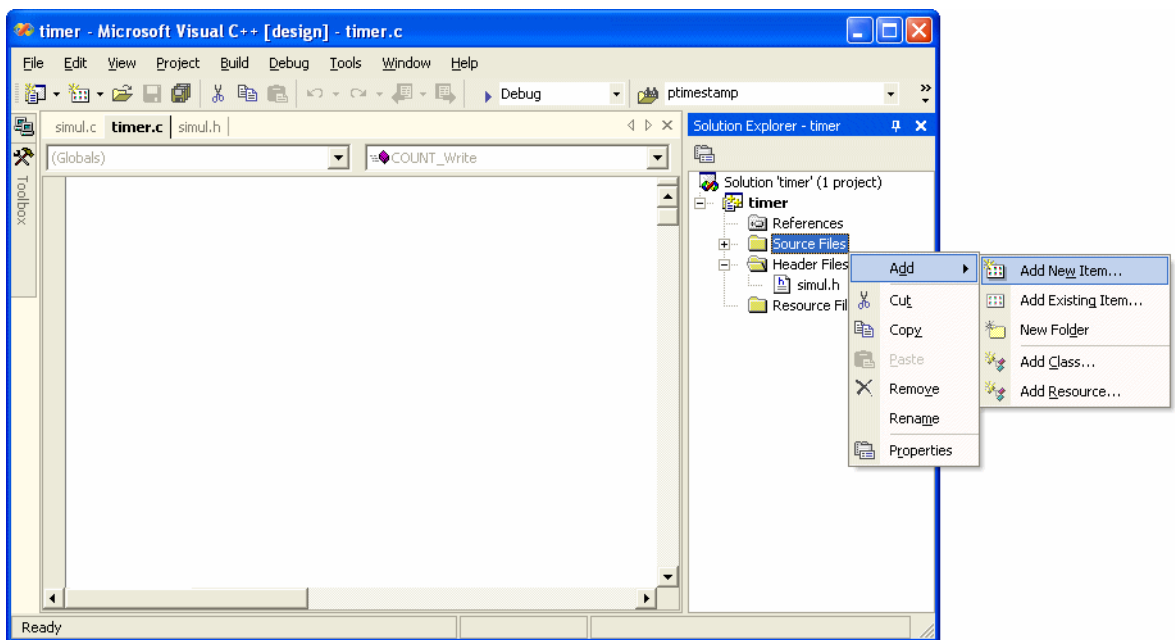
In 'Location' field, enter model location (in this case is a "C:\timer").

Click Ok button to proceed.

In "Win32 Application Wizard" window, click on 'Application Settings' then select DLL as 'Application type' and click Finish button.

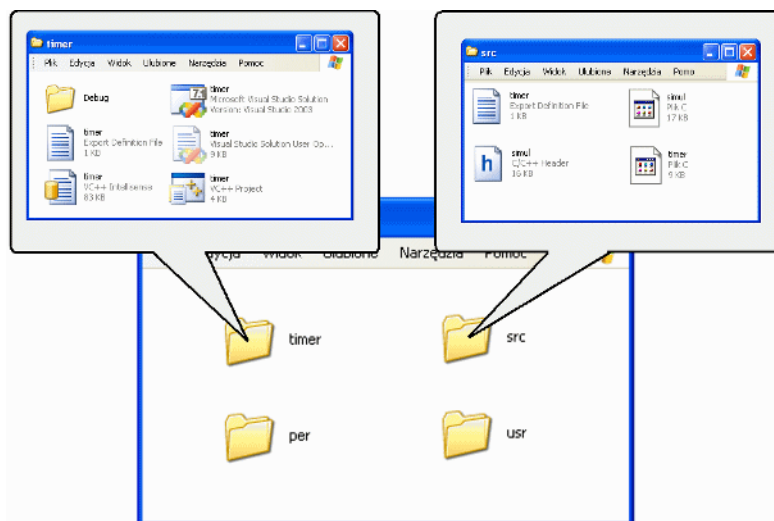


Once our programming environment is ready, we need to create a files structure of our model. The project consists of 4 source files: simul.h, simul.c and created by us files "model_name".c and "model_name".def

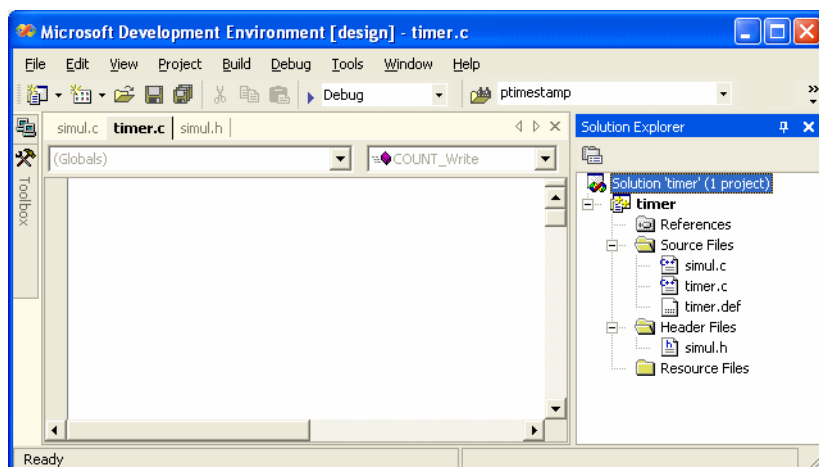


To 'Source Files' add: `simul.c`, `'model_name'.c` and `'model_name'.def` (Module Definition File). To Header files add: `simul.h`. Content `'model_name'.def` file should look like this:

LIBRARY	<code>simple_port</code>
DESCRIPTION	<code>'TRACE32 Hardware Simulation Model'</code>
EXPORTS	<code>SIMUL_Interface</code>



After adding all required files structure should look like this:



Source code listing

First what designer need to do, is to include a “simul.h” library, which contains all necessary functions. For better readability of the code, registers offsets should be defined as “**short_name**”_OFFSET. Base address should be also defined as “**module_name**”_BASE. “NUM_OF_REGS” definition specifies a number of registers in the module.

Example:

“Timer 0 module” ->	TIMER0_BASE
“Control register” ->	CTR_OFFSET

```
#include "simul.h"

#define NUM_OF_REGS 5
#define COUNT_OFFSET 0x0000
#define CTR_OFFSET 0x0004
#define SVAL_OFFSET 0x0008
#define EVAL_OFFSET 0x000c
#define IR_OFFSET 0x0010
#define TIMER0_BASE 0xFF000000
```

Type definition of 32 bits width register

```
typedef struct
{
    simulWord32    count, ctr, sval, eval, ir;
}
Regs;
```

```
typedef struct
{
    simulWord32 startaddress;
    int         bustype, reset, work, intport, chport;
    Regs        regs;
    simulTime    ctimestamp;
    void *       ptimer;
}
Timer;
```

Function declaration used to link registers offsets with module base address. Inside there are previously defined registries offsets. **SimulWord32** specifies variables type.

```
static simulWord32 regs_offset[NUM_OF_REGS] = {
    COUNT_OFFSET, CTR_OFFSET, SVAL_OFFSET, EVAL_OFFSET, IR_OFFSET
};
```

Function declaration used to write values in selected register. Variables have to be related to the module registers.

```
void * regwrite_func[NUM_OF_REGS] = {
    COUNT_Write, CTR_Write, SVAL_Write, EVAL_Write, IR_Write
};
```

Function declaration used to read values from selected register. Variables have to be related to the module registers.

```
void * regread_func[NUM_OF_REGS] = {  
    COUNT_Read, CTR_Read, SVAL_Read, EVAL_Read, IR_Read  
};
```

Registers initialization function. Each register is linked to module base address by "for" loop and calls two callback functions. First one is responsible for writing and second one is responsible for reading registers values.

```
static void Regs_Init(simulProcessor processor, Timer * timer)  
{  
    simulWord from, to;  
    int i;  
  
    for (i = 0; i < NUM_OF_REGS; i++)  
    {  
        from = timer->startaddress + regs_offset[i];  
        to = from + 3;  
  
        SIMUL_RegisterBusWriteCallback(processor, regwrite_func[i],  
(simulPtr) timer, timer->bustype, &from, &to);  
        SIMUL_RegisterBusReadCallback(processor, regread_func[i],  
(simulPtr) timer, timer->bustype, &from, &to);  
    }  
}
```

Write function for a particular register. This function have to be attributed to each register of access type as WRITE or READ/WRITE.

```
static int SIMULAPI COUNT_Write(simulProcessor processor, simulCallback-  
Struct * cbs, simulPtr private)  
{  
    Timer * timer = (Timer*) private;  
    SIMUL_InsertWord(processor, &timer->regs.count, 32, &cbs->x.bus.address,  
cbs->x.bus.width, &cbs->x.bus.data);  
    return SIMUL_MEMORY_OK;  
}
```

Read function for a particular registry. This function have to be attributed to each register of access type as READ or READ/WRITE.

Note: If register is visible in TRACE32 window, then function is called with each step.

```
static int SIMULAPI COUNT_Read(simulProcessor processor, simulCallbackStruct * cbs, simulPtr private)
{
    Timer * timer = (Timer*) private;
    SIMUL_ExtractWord(processor, &timer->regs.count, 32, &cbs->x.bus.address,
cbs->x.bus.width, &cbs->x.bus.data);
    return SIMUL_MEMORY_OK;
}
```

In write function, you can place a direct function of the register handle i.e. operations on the register are performed at the same cycle as at write to it. All operations are performed on variable such as “reg” and a result is assigned to the register.

```
/* CTR (Control Register) */

static int SIMULAPI CTR_Write(simulProcessor processor, simulCallbackStruct * cbs, simulPtr private)
{
    Timer * timer = (Timer*) private;
    simulWord32 reg;
    SIMUL_InsertWord(processor, &reg, 32, &cbs->x.bus.address,
cbs->x.bus.width, &cbs->x.bus.data);

    if (reg & 0x2) /* Timer reset */
    {
        timer->regs.count = 0x0;
        reg = 0x0;
        timer->regs.sval = 0x0;
        timer->regs.eval = 0x0;
    }
    if (((reg & 0x70)==0x20)||((reg & 0x70)==0x60))
    {
        timer->work=1;
    }
    timer->regs.ctr = reg & 0xffffffff;
    return SIMUL_MEMORY_OK;
}
```

Function returns "SIMUL_MEMORY_OK".

Timer handle function. The operations are executed with each step. Inside this function is implemented: increment / decrement value of the register, bit masks, ect.

For example, in place of stars may be located functions corresponding to the various modes of timer. By setting the corresponding bits in Control register, different operating modes are selected.

```
/* --- IntReq timer --- */

int SIMULAPI IntReqTimer(simulProcessor processor, simulCallbackStruct *
cbs, simulPtr private)
{
    Timer * timer = (Timer*) private;
    simulWord data;

    if (timer->regs.ctr & 0x1)                /* Timer enabled */
    {
        timer->regs.count++;
    }
    switch ((timer->regs.ctr >> 4) & 0x7)
    {
    case 0x0:
        *
        *
        *
    break;
        *
        *
        *
        *
    }

    return SIMUL_TIMER_OK;
}
```

Function returns "**SIMUL_TIMER_OK**".

Ports are used for communication between models. Each port has its own number and may take a state of 0 or 1. Port -2 enables interruption of processor. To enable interruption, set '1' on port, so then after a fulfillment of a condition, port will be on '1'.

In **Intport** variable stores interruption port number.

```
if ((timer->regs.ir))
{
    data = 1;
    SIMUL_SetPort(processor, timer->intport, 1, &data);
}
```

After interrupt handling is completed, exit from interrupt have to be done by clearing a state of port responsible for it.

```
/* IR (IR Register) */

timer->regs.ir &= ~(reg & 0xff);
    if (!timer->regs.ir)                /* deassert interrupt */
    {
        simulWord data = 0;
        SIMUL_SetPort(processor, timer->intport, 1, &data);
    }
```

Function responsible for a timer reset is called after model initialized. Because **TIMER_Reset** function is called after initialization, simulation timer (not timer PSM) can be started by **SIMUL_StartTimer** function. Depending on module needs, different controls flags can be used.

Presented timer model uses two flags **SIMUL_TIMER_REPEAT** and **SIMUL_TIMER_CLOCKS**.

SIMUL_GetClock function returns number of clock cycles since the beginning of the simulation, increasing on each step.

```
static int SIMULAPI TIMER_Reset(simulProcessor processor,
simulCallbackStruct * cbs, simulPtr private)
{
    Timer * timer = (Timer*) private;
    simulTime time, nowtime;
    memset(&timer->regs, 0x00, sizeof(timer->regs));
    timer->reset = 0;

    SIMUL_GetClock(processor, 0, &nowtime);
    time =1;
    SIMUL_StartTimer(processor, timer->ptimer,
SIMUL_TIMER_REPEAT | SIMUL_TIMER_CLOCKS, &time);

    return SIMUL_RESET_OK;
}
```

Function returns "**SIMUL_RESET_OK**".

The most important in simulation module is initialization function. Is called only once at the very beginning. All callback function here are placed. In presented timer model, **SIMUL_RegisterResetCallback** function responsible for a timer reset and **Regs_Init** function responsible for events handling in the registers, have been called. There are also declarations of variables. Because there are no global variables in a model, function **SIMUL_Alloc** is used.

```
int SIMULAPI SIMUL_Init(simulProcessor processor,
simulCallbackStruct * cbs)
{
    Timer * timer;
    int i;
    strcpy(cbs->x.init.modelname, __DATE__ " Timer Model");
    timer = (Timer*) SIMUL_Alloc(processor, sizeof(Timer));

    timer->bustype      = 0;
    timer->intport      = -2;
    timer->startaddress = TIMER0_BASE;

    Regs_Init(processor, timer);

    SIMUL_RegisterResetCallback(processor, TIMER_Reset, (simulPtr) timer);
    timer->ptimer = SIMUL_RegisterTimerCallback(processor, IntReqTimer,
(simulPtr) timer);
    TIMER_Reset(processor, cbs, timer);

    return SIMUL_INIT_OK;
}
```

Function returns "**SIMUL_INIT_OK**".

Each model can be called with parameters. Correct handling of model parameters must be ensured. The "i" variable specifies the number of checked parameters (depending on needs of simulation model). Presented timer is set (by default) to base address stored in **TIMER0_BASE** variable (under condition that as a first parameter is "timer" string), but if necessary it can be overwritten with any other value. Second parameter is the number of port (responsible for interrupt). By default value is stored in variable and is equal "-2" but if necessary it can be overwritten with any other value. Function at the end calls warning function, which is designed to notify module parameters.

```
for (i = 1; i <= cbs->x.init.argc - 1; i++)
{
    if (i == 1)
    {
        if (strstr(cbs->x.init.argv[1], "timer") != NULL)
            timer->startaddress = TIMER0_BASE;
        else
            timer->startaddress = cbs->x.init.argvport[1];

        continue;
    }

    else if (i == 2)
    {
        if (strstr(cbs->x.init.argv[2], "cpu") != NULL)
            timer->intport = SIMUL_PORT_INTERRUPT;
        else if (strstr(cbs->x.init.argv[2], "noport") == NULL)
            timer->intport = cbs->x.init.argvport[2];
        continue;
    }

    SIMUL_Warning(processor, "usage parameters: [<base address|name>
<interrupt port>]");
    return SIMUL_INIT_FAIL;
}
```

Initialization scripts

Following script provides settings for TRACE32 environment. The script contains a set of commands to run timer with initial settings such as work mode, initial and final value of incremented register, interrupt options, etc. The script also contains commands for opening and distribution of appropriate windows in TRACE32.

The script should look like this:

Header of script contains such data as module name, developer, major functions, etc.

```
; -----  
; 'TIMER' Model Initialization Script  
; DATA (NICK)  
;  
; A brief comment on the specific characteristics (ports) [not required]  
; -----
```

SIM.UNLOAD command is used to clean the memory configuration for a simulation model. After **SIM.LOAD** command should be a path to DLL library that contains implemented module.

```
; --- Timer initialization script ---  
  
sim.unload  
  
sim.load ../timer/Debug/timer.dll "timer" "cpu"  
per ../per/timer.per  
  
SYStem.Up  
  
d.l
```

This sequence allows to set model in infinite loop, after 3 steps.

```
; --- processor infinite loop ---  
d.a 0x0 nop  
d.a 0x4 nop  
d.a 0x8 b $ - 8; infinite loop
```

When interrupt event occurred, interrupt have to be handled and exit. At the time of interrupt event, program jumps to 0x18 address where is procedure to clear interrupt. In next step program return to an infinite loop.

```
; --- interrupt service routine ---
d.a 0x18 str r0, [r1, #0x10]; clear interrupt flag
d.a 0x1c subs pc, r14, #4; return from interrupt
```

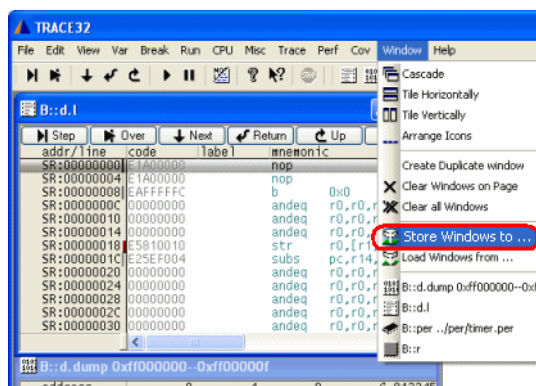
Configuration for ARM core. Set program counter to 0x0 value. The register r0 holds value to be entered in the “interrupt register”. The register r1 holds timer base address.

```
; --- ARM core configuration ---
r.s pc 0x0; set program counter to 0x0
r.s r0 0x1; value to write to register
r.s r1 0xff000000; base address of Timer
```

Configure timer registers. Set timer to appropriate operating mode determines the initial and final value (depending on the mode) and begins counting.

```
; --- Timer configuration ---
d.s d:0xff000004 %l 0x59; timer enable,loop,from sval to eval
d.s d:0xff000008 %l 0x00; start value
d.s d:0xff00000c %l 0x20; end value
r.s I 0; enable interrupts
b.s 0x18; set breakpoint on beginning of interrupt
```

For better visibility, set windows position in TRACE32. The easiest way to do it is deploy windows as intended and then save the settings by clicking on the bookmark **Window / Store Windows to...** Initialized model has a preview of all functions.



Windows position settings can be copy to the initialization script.

```
B::

TOOLBAR ON
STATUSBAR ON
WINPAGE.RESET

WINCLEAR
WINPOS 0.0 19.154 70. 4. 15. 1. W001
d.dump 0xff000000--0xff00000f

WINPOS 0.0 0.0 56. 14. 15. 1. W002
WINTABS 10. 10. 25. 62.
d.l

WINPOS 58.857 12.077 47. 13. 0. 0. W000
per ../per/timer.per

WINPOS 60.143 0.23077 46. 9. 0. 0. W003
r

WINPAGE.SELECT P000

enddo
```

