Dipartimento DII
Università Politecnica delle Marche (Italy)

# PKtool 2.3.0

Framework extension for transaction level power analysis

## 1 INTRODUCTION

This documentation is an appendix of the PKtool user manual, which deals with the application of PKtool for power estimations at transaction level. At the present time, PKtool makes available specific capabilities for systems modelled through the TLM 2.0 subset of the SystemC language (SystemC/TLM). In this documentation, the term *PKtool/TLM* will be used to indicate the transaction level extension of PKtool, whereas the basic framework, i.e. the components and functionalities strictly described in the user manual, will be referred to as *basic PKtool*.
PKtool/TLM enhances the basic PKtool functionalities and provides the means to configure a SystemC/TLM description for power analysis. The implementation of PKtool/TLM is based on a class library included in the namespace *pk_tlm*. Such namespace depends on the components of SystemC/TLM and is merged with the namespaces constituting basic PKtool. The inclusion of the pk_tlm namespace in the building of the PKtool library can be enabled or disabled by the user, such that he can choose whether to make operative PKtool/TLM or just the basic capabilities.
PKtool/TLM has been realized on the basis of the simulation semantics and modeling constructs related to SystemC/TLM. In particular, the main entities considered for power estimations are the interface functions used to model transaction level communications. PKtool/TLM power analysis are mainly based on the association between such functions and suitable power models. The power models available for PKtool/TLM applications are included in the PKtool default energy library. Their use is restricted only to PKtool/TLM applications.
The next sections illustrate the modalities to realize power analysis in PKtool/TLM environment, with the support of concrete examples. However, a proper comprehension requires the knowledge of the basic PKtool framework and, more specifically, the contents discussed in the PKtool user manual. The arguments exposed in this documentation are organized as follows:

| | |
|---|---|
| Section 2: | Overview of SystemC/TLM |
| Section 3: | Conceptual aspects in PKtool/TLM analysis |
| Section 4: | Preliminary configuration steps |
| Section 5: | Reference examples |
| Section 6: | Selection of TLM functions |
| Section 7: | Power model specification |
| Section 8: | Configuration file |
| Section 9: | Analysis results |
| Section 10: | Available power models |

## 2 OVERVIEW OF SYSTEMC/TLM

Before entering in the details of PKtool/TLM, it is convenient to have a preliminary overview on the SystemC/TLM language. This section does not provide a comprehensive report, but is focused only on those features relevant for PKtool/TLM applications. More detailed information can be found in the official SystemC/TLM documentation.
SystemC/TLM is an extension of the SystemC language which allows a high level modeling of the communications between interacting modules. In particular, the communication granularity is based on transaction objects. A transaction can be regarded as a consistent data amount exchanged by interacting modules and handled in atomic way. The transmission of a transaction is not fragmented by clock synchronization, but usually takes place in a single step with the possible specification of the processing time.
Within a SystemC/TLM representation, the links between modules do not consist in wire-like connections but are emulated by functional transport interfaces targeted to transactions handling. A single transaction is issued by means of a specific function call between a transmitter and a receiver module, more commonly referred to as *initiator* and *target*. An initiator is a module that can initiate a transaction, which means to create transaction objects and transmit them by calling specific interface functions. On the other hand, a target module is the final destination of a transaction. The physical path between an initiator and a target could eventually comprise interconnect units, such as arbiters or routers, that receive and issue again interface function calls (Fig. 1). In the transport of different transactions, the role of a module may be variable and cover the function of initiator, target or interconnect unit in alternate way.

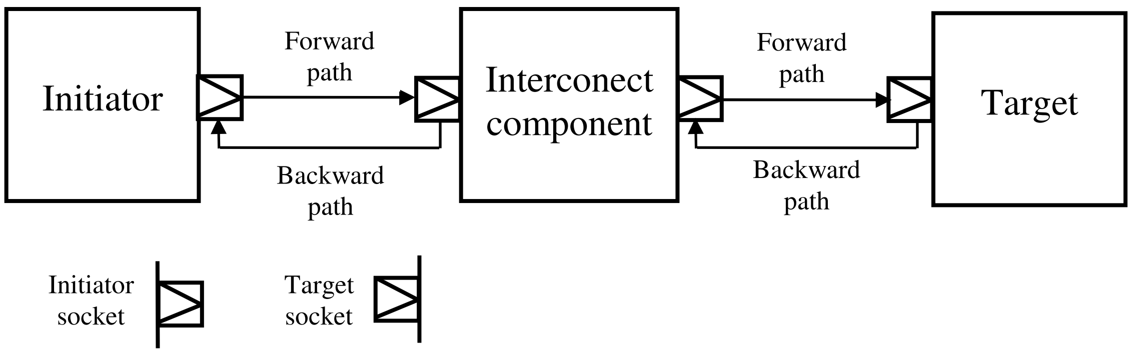

Figure 1

SystemC/TLM provides several transport interfaces corresponding to different modalities to handle a transaction:

tlm_blocking_transport_if

tlm_fw_nonblocking_transport_if

tlm_bw_nonblocking_transport_if

tlm_fw_direct_mem_if

tlm_bw_direct_mem_if

tlm_transport_dbg_if

The virtual functions associated to such interfaces (TLM functions) are respectively the following:

```
void   b_transport (TRANS& trans, sc_core::sc_time& t)

tlm_sync_enum  nb_transport_fw (TRANS& trans, PHASE& phase, sc_core::sc_time& t )

tlm_sync_enum   nb_transport_bw (TRANS& trans, PHASE& phase, sc_core::sc_time& t)

bool   get_direct_mem_ptr (TRANS& trans, tlm_dmi& dmi_data)

void  invalidate_direct_mem_ptr (sc_dt::uint64 start_range, sc_dt::uint64 end_range)

unsigned   int transport_dbg (TRANS& trans)
```

The input parameter *TRANS* is referred to a transaction object containing the relevant data to be transmitted; *phase* indicates the transaction phase when using the non-blocking interfaces, that allow to split a transaction into more TLM function calls between an initiator and a target module; *tlm_dmi* comprises additional attributes specific for direct memory access; *tlm_sync_enum* is the return type related to non-blocking transport interfaces, which reports the operative status after a TLM function call.

Transaction objects may be transported along two possible directions, from initiator to target and from target to initiator. Such directions are referred to as *forward path* and *backward path*, entailing inverse roles of an initiator and a target as caller or callee of a TLM function. Each TLM function is associated only to one of such paths and can transport transaction objects only in compliance with the specific direction. b_transport, nb_transport_fw and get_direct_mem_ptr are related to forward path, whereas nb_transport_bw and invalidate_direct_mem_ptr are related to backward path.

SystemC/TLM defines a standard transaction template called *generic payload* and immediately usable to create transaction objects. The TRANS parameter in the prototype of a TLM function is usually assigned to generic payload as default transaction type. Generic payload has been conceived to support a high interoperability for TLM models coming from different parties. Its features are explicitly related to memory-mapped bus protocols, including some typical attributes such as data, address and command. Nonetheless, it is possible to extend the generic payload also for modelling protocols other than memory-mapped buses.

TLM functions are aimed to define the modalities to transport and process transaction objects. The specific implementation of a TLM function must be reported in a module acting as callee, which can cover the role of target or initiator depending on the function. The calls to this function can be issued by a caller module that is connected to the callee module.

In order to facilitate the inter-module connections via transport interfaces, SystemC/TLM makes available ad-hoc components called *sockets*. A socket can be considered as an extension of the port concept, allowing a directly link between functional interfaces and interacting modules. Like a traditional port, a socket is instanced as a module member and can be bound to external modules according to specific rules. One of the primary use of a socket is as access point to call TLM functions from a caller module. SystemC/TLM provides two main socket types, namely *tlm_initiator_socket* and *tlm_target_socket*. They can be instanced respectively in an initiator and a target module, allowing to call the TLM functions respectively related to forward path and backward path.

## 3 CONCEPTUAL ASPECTS IN PKTOOL/TLM ANALYSIS

### 3.1 Estimation approach

As happens in basic PKtool analysis, PKtool/TLM provides power estimations at the level of the single modules constituting a monitored system. The modalities followed by PKtool/TLM to compute power estimations are strictly connected to the semantics of SystemC/TLM in the modeling of transaction-based communications. In particular, the main entities considered are the TLM functions that realize transaction transport.
Within a SystemC/TLM representation, the dynamic behaviour of a module is dependent on the transactions occurred during a simulation. This is because transaction handling determines the operations connected to the I/O evolution. In conceptual terms, a transaction entails a specific energy dissipation that can be conveniently split onto the modules involved in the transaction (the initiator, the target and possible interconnect units). Each of these modules is so concerned with an its own energy dissipation due to the transaction. In this way, with regard to the dynamic behaviour, the overall energy dissipation may be estimated by summing up all the contributions due to the transactions occurred during the simulation.
Since the handling of a transaction is mapped onto the TLM functions concerning the involved modules, monitoring these functions is a key factor to determine the dissipation contributions of each module. To this end, PKtool/TLM is able to sample the calls to the TLM functions related to a transaction and allows to estimate an energy dissipation for each module involved in such function calls. More precisely, each module can be associated to a set of power models that are linked to the TLM functions regarding the module. Every time a TLM function is called, an energy dissipation is estimated for the caller and callee modules, by computing the respective power models linked to such function (Fig. 2). Then, such contributions are summed up to the overall energy estimations held for each module.
In general, including the possible presence of interconnect units, a transaction occurrence may entail several power estimations, one for each module involved in the transaction. In the simplest case of a transaction concerning only an initiator and a target, at most two power estimations could be computed.

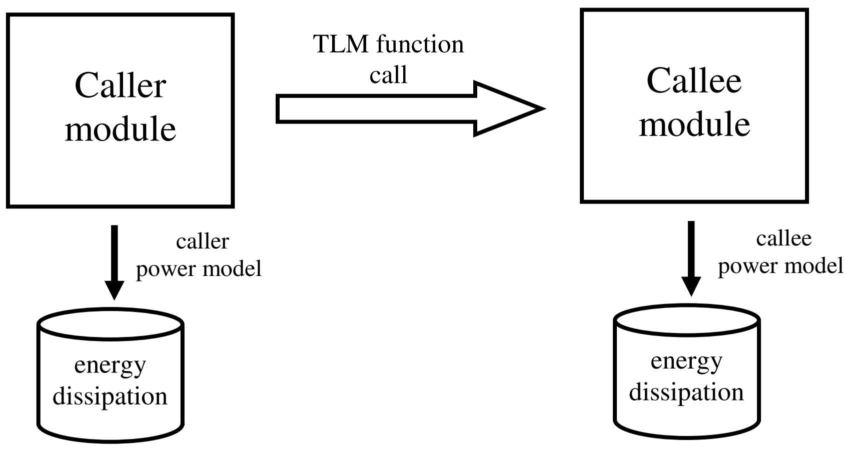

Figure 2

## 3.2 Formulas applied in energy estimations

According to the approach previously depicted, the basic formula used to estimate the energy dissipation of a module is given by the following expression:

$$E_{mod} = \sum_{i=1}^{N} e_i \qquad (1)$$

where N is the number of transactions occurred during a simulation and $e_i$ stands for the energy estimation of a single transaction. $e_i$ is referred to the call of a particular TLM function that may concern the module in the role of caller or callee in the considered transaction.
Equation (1) represents the basic formula used by PKtool/TLM in its estimations and in many cases may be suitable to achieve an acceptable accuracy. However, it is possible to consider a more refined expression, capable to include also the energy dissipation in idle states:

$$E_{mod} = PT_{idle} + \sum_{i=1}^{N} e_i \qquad (2)$$

The second contribution corresponds to the transaction-related dissipation, as reported in (1), whereas the first contribution is referred to the idle dissipation. $T_{idle}$ is the overall idle time and P the power dissipated in idle states. In general, the run-time behaviour of a module may be constituted by an alternation between active states (in which transactions are handled) and idle states (Fig. 3). Idle states are not covered by TLM functions and their dissipation can be properly modelled by static power dissipation.

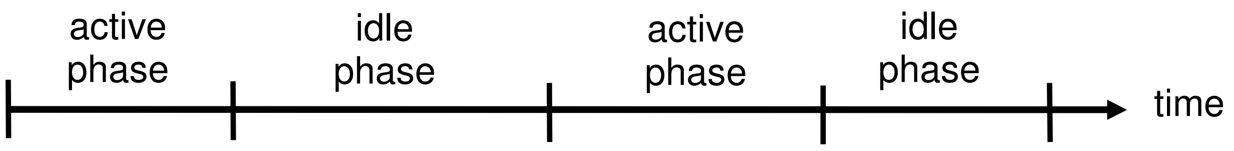

Figure 3

A possible approach to determine the energy costs $e_i$ could consist in a macro-modeling procedure on a low-level representation of the system, e.g. a gate-level model. This procedure is usually based on accurate energy dissipation measures carried out on the low-level model and referred to complete executions of the TLM functions. The energy costs of each TLM function can be extracted from these low-level measures.
In the simplest case, a TLM function may be associated to a unique energy cost, which is assumed as its energy dissipation every time the function is called. Nonetheless, more accurate estimations may be achieved by defining a wider set of energy costs that can be associated to the characteristic data of the function (TLM data). Such data are given by the input parameters and return value of the function, and often allow to specify different operative conditions. The extraction of these more refined costs could require a more complex macro-modeling procedure, in which the low-level measures are to be carried out for the different operative conditions specifiable by TLM data. As will be illustrated in section 10, PKtool/TLM provides power models that permit to associate a

TLM function to a unique energy costs as well as to several energy costs for the possible values of the TLM data.
In regard to idle dissipation, the static power P must be specified at the beginning of a PKtool simulation. $T_{idle}$ is automatically determined by PKtool/TLM, considering the average execution times of the TLM functions that may concern the module. More specifically, $T_{idle}$ is evaluated by PKtool through the formula:

$$T_{idle} = T - \sum_{i=1}^{N} t_i \qquad (3)$$

where T is the overall simulation time and the $t_i$ terms represent the execution times of the TLM functions called during the simulation. In order to evaluate equation (3), an average execution time must be specified for each TLM function concerning the module. Like P and $e_i$, such average time is communicated by the user at the beginning of a PKtool/TLM simulation.
Within a PKtool/TLM analysis, the output energy estimations can be determined by applying simply equation (1) or the more accurate version given by (2). Actually, this choice may depend on the adopted technology mapping. In the most advanced microelectronic technologies, static power dissipation is often not negligible such that the contribution due to idle states may be relevant. A user can choose the formula to be applied through a configuration setting that specifies whether or not to include the idle dissipation.

## 4 PRELIMINARY CONFIGURATION STEPS

The application of PKtool/TLM requires some preliminary steps to make operative the related components and functionalities. This section deals only with those interventions to be done at code level; all the compilation environment settings are explained in the INSTALL file placed in the top directory of a PKtool release.
First of all, it is necessary to enable the PKtool/TLM framework when building the PKtool library. PKtool/TLM can be enabled or disabled through the macro PK_TLM_ENABLED, defined in the header file pk_settings.h inside the directory src/PKtool/kernel. For enabling/disabling PKtool/TLM, the user must uncomment/comment this macro. By default, PKtool/TLM is not enabled.
The next step consists in making visible the PKtool/TLM class library. This intervention is realized by including the PKtool header file, *PKtool.h*, before the instance of PKtool components. In practice, the rules to follow would be the same valid for basic PKtool applications and illustrated in the user manual (section 5). The only difference is to guarantee that the SystemC/TLM header, *tlm.h*, is reported before the PKtool header. Such requisite is necessary to ensure a correct compilation, since several PKtool/TLM components depend on the contents in tlm.h.

## 5 REFERENCE EXAMPLES

The configuration of a SystemC/TLM description for PKtool/TLM analysis will be shown through a concrete application. More specifically, we will consider one of the illustrative examples included in SystemC/TLM releases, namely *at_2_phase*. The related code and documentation are available in the directory *examples/tlm/at_2_phase*.
At_2_phase is a multi-device system based on the following architecture:

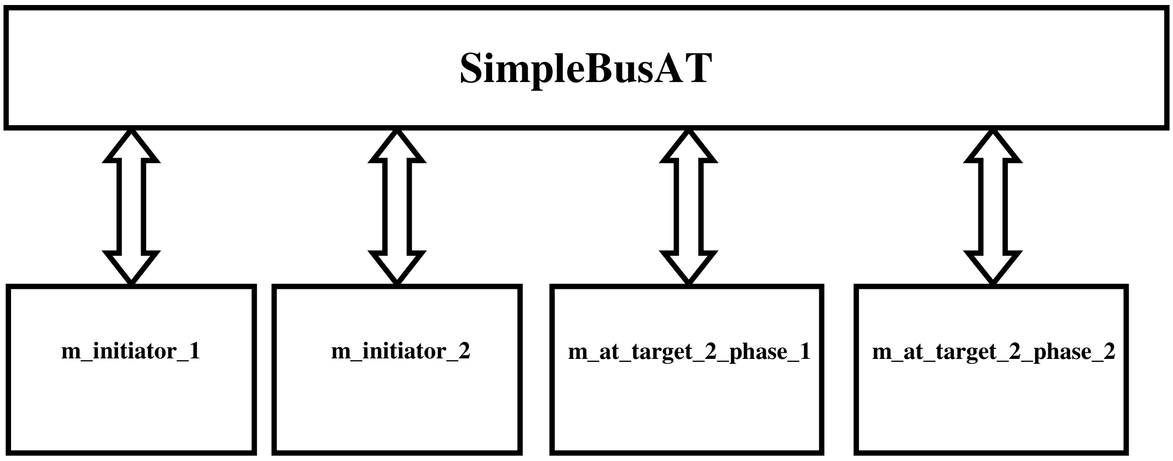

Figure 4

where we can distinguish two initiators (m_initiator_1, m_initiator_2) and two targets (m_at_target_2_phase_1, m_at_target_2_phase_2) connected through a 2x2 router. The system functionality consists in communication tasks based on the non_blocking transport interface and carried out via the TLM functions nb_transport_fw and nb_transport_bw. The target module is a top-level module whereas the initiator module is wrapped in a more complex unit called initiator_top. Initiator and target modules are respectively associated to the classes select_initiator and at_target_2_phase.
An individual module is configured for PKtool/TLM analysis by following some systematic steps; these latter will be shown with reference to the initiator and target modules in at_2_phase. Such steps are mostly realized by defining and instancing power_module counterparts, with the same modalities valid for basic PKtool applications. The power_module classes for the initiator and target modules are defined below, considering only the essential details:

```
#include "tlm.h"        // SystemC/TLM header
#include "select_initiator.h"
#include "at_target_2_phase.h"

#include "PKtool.h"     // PKtool header

// power_module class for the initiator module

POWER_MODULE_CLASS(select_initiator)
{

 PK_USES_ENERGY_MODELS
 PK_HAS_PROCESS(select_initiator);


 // constructor

 POWER_MODULE(select_initiator)
      ( sc_core::sc_module_name name
       , const unsigned int  ID
       , sc_core::sc_time end_rsp_delay
      ):
     select_initiator(name,ID,end_rsp_delay), PK_PMB_CTOR
     { }

};


// power_module class for the target module

POWER_MODULE_CLASS(at_target_2_phase)
{

 PK_USES_ENERGY_MODELS

 PK_HAS_PROCESS(at_target_2_phase);

 // constructor

 POWER_MODULE(at_target_2_phase)
      ( sc_core::sc_module_name  module_name,
        const unsigned int       ID,
        const char               *memory_socket,
        sc_dt::uint64            memory_size,
        unsigned int             memory_width,
        const sc_core::sc_time   accept_delay,
        const sc_core::sc_time   read_response_delay,
        const sc_core::sc_time   write_response_delay
      ):
      at_target_2_phase(module_name,ID,memory_socket,
                         memory_size, memory_width,accept_delay,
                        read_response_delay, write_response_delay),
      PK_PMB_CTOR
     { }

};
```

In the power_module classes the macro PK_USES_ENERGY_MODELS must be reported to guarantee the correct linkage to the power model library. The constructor definitions follow the rules explained in the PKtool user manual (section 6.3).
Once defined the power_module classes, it is possible to convert initiator and target modules into power_module counterparts. In our example, we will select the first initiator and the first target in

the layout of Fig. 4. Both of these modules are created statically; the target is instanced through this instruction in the top-level scope:

```
at_target_2_phase    m_at_target_2_phase_1;
```

The power_module conversion is achieved by including the original type in the macro POWER_MODULE:

```
POWER_MODULE(at_target_2_phase)   m_at_target_2_phase_1;
```

As concerns the initiator, the related instance instruction is wrapped in the initiator_top class:

```
class initiator_top
  : public sc_core::sc_module
{
 // class body
 ...
 ...

 select_initiator     m_initiator;

}
```

The power_module conversion is realized in the same way seen for the target:

```
class initiator_top
  : public sc_core::sc_module
{
 // class body
 ...
 ...

 POWER_MODULE(select_initiator)   m_initiator;

}
```

At this point, initiator and target modules are selected for PKtool/TLM analysis. Actually, when launching a PKtool/TLM simulation, two select_initiator power_modules are enabled, as a consequence of the indirect instances from the top-level modules m_initiator_1 and m_initiator_2. In the considered example, the power_module in m_initiator_2 will be disabled in the specification procedure at the beginning of the PKtool/TLM simulation.
In the at_2_phase system the overall simulation time is not explicitly declared in the sc_start function, which is reported without a time reference. For a suitable calculation of the overall idle time and average power dissipation, in the sc_main function we have specified the instructions:

```
sc_core::sc_time sim_time(3800, sc_core::SC_NS);
pk_set_simtime(sim_time);
```

as explained in the PKtool user manual (Section 10), such instructions allow to set a simulation time considered by PKtool in the estimation tasks. In this example, we have assumed a simulation time equal to 3800 ns.

## 6 SELECTION OF TLM FUNCTIONS

When configuring a module for PKtool/TLM analysis, the first operation is to select the TLM functions involved in transaction handling. The selection of a TLM function consists of two possible alternatives, depending on whether the module covers the role of caller or callee. If the module acts as callee, this means a specific function implementation is reported in the module class. In this case, the TLM function is selected for PKtool/TLM estimations by specifying a macro instruction in the power_module class. Such instruction depends on the function according to the following associations:

| TLM FUNCTION | SELECTION MACRO |
|---|---|
| b_transport | PK_B_TRANSPORT_2 |
| nb_transport_fw | PK_NB_TRANSPORT_FW_2 |
| nb_transport_bw | PK_NB_TRANSPORT_BW_2 |
| get_direct_mem_ptr | PK_GET_DIRECT_MEM_PTR_2 |
| invalidate_direct_mem_ptr | PK_INVALIDATE_DIRECT_MEM_PTR_2 |

As an example, let us consider the target module in the at_2_phase system. This unit implements the transport interface tlm_fw_transport_if and so may behave as callee for the functions b_transport, nb_transport_fw and get_direct_mem_ptr. Actually, the module functionality is reproduced only by nb_transport_fw, which is defined through a meaningful implementation; the other functions are defined in the module class through dummy implementations. Regardless this distinction, in our example let us suppose we want to select nb_transport_fw and get_direct_mem_ptr. For this purpose, these instructions must be specified in the power_module class:

```
POWER_MODULE_CLASS(at_target_2_phase)
{

  //implementation code
  ...
  ...

  // instructions for selecting nb_transport_fw and get_direct_mem_ptr

  PK_NB_TRANSPORT_FW_2
  PK_GET_DIRECT_MEM_PTR_2

};
```

In this way, the two functions are automatically scheduled for the next configuration phase consisting in power model association.
An important prerequisite for the correct compilation of the previous instructions is the public or protected access of the selected functions. If one of such functions were declared private in the module class, a compilation error could probably be raised. For this reason, a specific modification has been necessary in the at_target_2_phase class, in which the function get_direct_mem_ptr is

originally declared private. In order to make possible its selection, the access to this function has been made protected.
If the module acts as caller with regard to a TLM function, the selection procedure is slightly more articulated because it is also necessary to indicate the socket used as access point to the function. In general, a caller module can comprise several socket objects related to a same transport interface. Each of these sockets may be connected to an external module that provides a specific implementation for the function. As a consequence, the TLM function can be properly selected only if the corresponding socket is specified. All that entails a selection procedure in which the user must indicate first the socket and then the function. The socket is specified with the same rules to instance an augmented port. In concrete terms, the original type of the socket must be replaced by a matching augmented type provided by PKtool/TLM. For example, let us consider this socket object instanced in the initiator module:

```
class select_initiator :
        public sc_core::sc_module,
        virtual public tlm::tlm_bw_transport_if<>
{
 // implementation code
   ...

   tlm::tlm_initiator_socket<>     initiator_socket;

}
```

The socket is called *initiator_socket* and is associated to the socket type *tlm_initiator_socket,* with the template parameters assigned to default values. In order to select this socket, the instance instruction must be modified in this way:

```
pk_tlm::tlm_initiator_socket_aug<>     initiator_socket;
```

where the original type has been replaced with the corresponding augmented type *pk_tlm::tlm_initiator_socket_aug*. At the present time, PKtool/TLM makes available the augmented counterparts for the sockets belonging to the TLM-2.0 interoperability layer, i.e. *tlm_initiator_socket* and *tlm_target_socket.*
At this point, initiator_socket has become an augmented socket and this makes it possible to select the TLM functions called through it. The pertinent instructions are based on parameterized macros that depend on the specific function. The below scheme shows the correspondences between TLM functions and selection macros:

| **TLM FUNCTION** | **SELECTION MACRO** |
|---|---|
| b_transport | PK_B_TRANSPORT_1(socket_name) |
| nb_transport_fw | PK_NB_TRANSPORT_FW_1(socket_name) |
| nb_transport_bw | PK_NB_TRANSPORT_BW_1(socket_name) |
| get_direct_mem_ptr | PK_GET_DIRECT_MEM_PTR_1(socket_name) |
| invalidate_direct_mem_ptr | PK_INVALIDATE_DIRECT_MEM_PTR_1 (socket_name) |

The macro parameter *socket_name* stands for the socket by which the function is called. Such macro instructions must be reported in the constructor of the power_module class.
In our example, the functions that can be called via initiator_socket are b_transport, nb_transport_fw and get_direct_mem_ptr. These functions are the same implemented by the target module in the role of callee. If we want to select b_transport and nb_transport_fw, we must report the following instructions in the constructor of the select_initiator power_module:

```
POWER_MODULE_CLASS(select_initiator)
{

 // implementation code
   ...
   ...

   // constructor

     POWER_MODULE(select_initiator)
        ( sc_core::sc_module_name name
         , const unsigned int  ID
         , sc_core::sc_time end_rsp_delay
         ):
      select_initiator(name,ID,end_rsp_delay), PK_PMB_CTOR
      {

       // instructions to select TLM functions for
       // initiator_socket

       PK_B_TRANSPORT_1(initiator_socket)
       PK_NB_TRANSPORT_FW_1(initiator_socket)
      }

}
```

At this point, the two functions are automatically scheduled for the next configuration step, i.e. power model association.
Developing further the configuration of the select_initiator module, in addition to the functions called via socket, it is also possible to select functions implemented by the module and handled in the role of callee. In this case, such functions are related to the transport interface tlm_bw_transport_if, and are given by nb_transport_bw and invalidate_direct_mem_ptr. If we want to select nb_transport_bw, the corresponding macro instruction must be reported in the power_module class:

```
POWER_MODULE_CLASS(select_initiator)
{

  // implementation code
   ...
   ...

  PK_NB_TRANSPORT_BW_2
}
```

In consequence of the access constraint previously explained, a specific modification has been necessary in the select_initiator class. In the original code, the function nb_transport_bw is declared private; in order to make possible its selection, the access to this function has been made protected.
The configuration procedure allows to include also the energy dissipation in idle states. As explained in section 3, such contribution takes into account the dissipation in the inactivity periods, in which no transaction is handled. The inclusion of the idle dissipation is optional and is enabled by specifing the macro PK_TLM_ENABLE_IDLE_ESTIM in the power_module constructor:

```
// constructor

     POWER_MODULE(select_initiator)
        ( sc_core::sc_module_name name
         , const unsigned int  ID
         , sc_core::sc_time end_rsp_delay
         ):
      select_initiator(name,ID,end_rsp_delay), PK_PMB_CTOR
      {

       // constructor code
       ...
       ...
        PK_TLM_ENABLE_IDLE_ESTIM

      }
```

Briefly summarizing, the energy dissipation of a module can be characterized through three distinct contributions: the transaction dissipations due to implemented and socket-called functions, and the idle dissipation. As shown in this section, each of these contributions is selected for PKtool/TLM analysis through specific instructions. The configuration of select_initiator module has been set up to comprise all these components, thus providing a reference to the most complete case. For the sake of clarity, we report the whole definition of the related power_module class:

```
POWER_MODULE_CLASS(select_initiator)
{

 PK_USES_ENERGY_MODELS

 PK_HAS_PROCESS(select_initiator);


 // constructor

 POWER_MODULE(select_initiator)
 ( sc_core::sc_module_name name,
   const unsigned int  ID,
   sc_core::sc_time end_rsp_delay
  ):
  select_initiator(name, ID, end_rsp_delay), PK_PMB_CTOR
  {

   PK_B_TRANSPORT_1(initiator_socket)
   PK_NB_TRANSPORT_FW_1(initiator_socket)

   PK_TLM_ENABLE_IDLE_ESTIM
  }

  PK_NB_TRANSPORT_BW_2

}
```

## 7 POWER MODEL SPECIFICATION

After selecting the TLM functions to be monitored, the next step is the specification of a power model for each of these functions. This association is realized through the same modalities valid in basic PKtool analysis, which consist in a multi-option routine at the beginning of a simulation. Such specification procedure concerns those modules that have been converted into power_module counterparts. The details of this step will be illustrated by continuing the configuration example for the at_2_phase system.
When a SystemC/TLM simulation is launched on at_2_phase, automatically also a PKtool/TLM simulation is started. At the beginning, this text appears on the command prompt window:

```
-------------------------------------------

    POWER_MODULE: top.m_at_target_2_phase_1

-------------------------------------------


OPTIONS FOR SPECIFYING THE POWER MODELS


1: interaction with window

2: reading from configuration file

3: no monitoring


select an option (1, 2, or 3) =
```

The headline reports the complete name of the first power_module, in this case referred to the target module m_at_target_2_phase_1. All the data communicated will be referred to such power_module. The meaning of the three options is the same of a basic PKtool simulation and is widely explained in the user manual (section 7).
If the option 1 is chosen, this text is displayed:

```
                POWER MODEL SPECIFICATION FOR TLM FUNCTIONS

available TLM power models: 5
related indexes: 51, 52, 53, 54, 55

                         IMPLEMENTED TLM FUNCTIONS

TLM functions selected: nb_transport_fw, get_direct_mem_ptr

function: nb_transport_fw
power model =
```

The first lines are information about the number of available TLM power models and their numeric identifiers. The current implementation of PKtool/TLM makes available TLM power models based on energy estimations and incorporated into the default energy library.
First of all, the user must specify the power models for the TLM functions implemented by the module, and concerning the role of callee. These functions are nb_transport_fw and get_direct_mem_ptr, in accordance with the selection shown in the previous section. In the request sentence the user is asked to specify the power model associated to nb_transport_fw. In this example, we could apply the power model identified by the index 51. This model is called TLM_1 and provides energy estimations based on a constant energy cost. Every time the TLM function is called, its energy estimation is given by such energy cost.
After reporting the index 51 in the request sentence, the user is asked to provide the energy cost:

```
power model: TLM_1    numeric index: 51
energy (nJ) =
```

The value must be expressed in nanojoules. In this example, we might assume 250 nJ.
Thereafter, this task is repeated for the function get_direct_mem_ptr:

```
function: get_direct_mem_ptr
power model =
```

Also for this function we might apply the TLM_1 power model, assuming an energy cost equal to 180 nJ.
At this point, we have defined the power models for all the TLM functions selected for m_at_target_2_phase_1, thus completing the configuration of this module.
The procedure now described will be repeated for all the other power_module instanced. In particular, the successive specification concerns the power_module wrapped by m_initiator_1, i.e. m_initiator_1_phase:

```
--------------------------------------------

    POWER_MODULE: top.m_initiator_1.m_initiator_1_phase

--------------------------------------------

OPTIONS FOR SPECIFYING THE POWER MODELS


1: interaction with window

2: reading from configuration file

3: no monitoring


select an option (1, 2, or 3) =
```

Also in this case, let us choose the specification option 1.

POWER MODEL SPECIFICATION FOR TLM FUNCTIONS

available TLM power models: 5
related indexes: 51, 52, 53, 54, 55

IMPLEMENTED TLM FUNCTIONS

TLM functions selected: nb_transport_bw

function: nb_transport_bw
power model =

m_initiator_1_phase implements only the function nb_transport_bw. Such function might be associated to the power model identified by the index 53. This model is called TLM_3 and provides energy estimations depending on transaction phase. Every time the TLM function is called, the energy estimation is determined considering the tlm_phase input parameter. tlm_phase is referred to an enumeration with four possible values: BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP. TLM_3 power model requires to associate an energy cost to each of these values:

power model: TLM_3  numeric index: 53
BEGIN_REQ energy (nJ) =

END_REQ energy (nJ) =

BEGIN_RESP energy (nJ) =

END_RESP energy (nJ) =

The phases BEGIN_REQ and END_RESP cannot be enabled when nb_transport_bw is called. As a consequence, the energy costs for such phases should be assigned to 0 nJ. The energy costs for END_REQ and BEGIN_RESP could be associated to 200 nJ and 250 nJ respectively.
The TLM functions selected for the initiator module include also two functions called via a socket port. The specification procedure continues considering the socket-related functions:

SOCKET-RELATED FUNCTIONS

SOCKET: initiator_socket
TLM functions selected: b_transport, nb_transport_fw

function: b_transport
power model =

The first text lines report the socket port (initiator_socket) and the associated functions (b_transport and nb_transport_fw). For each of this function, the user must specify a power model that will be evaluated every time the function is called via initiator_socket.
In this example, we could associate b_transport to the TLM_1 power model, with an energy cost equal to 120 nJ. As concerns nb_transport_fw, we could apply the power model identified by the index 54. This latter is called TLM_4 and provides energy estimations depending on transaction

status. Every time the TLM function is called, the energy estimation is determined on the basis of the tlm_sync_enum return value, which defines the transaction status. tlm_sync_enum is an enumeration with three possible values: TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED. The power model requires to associate an energy cost to each of these status values:

```
power model: TLM_4  numeric index: 54
TLM_ACCEPTED energy (nJ) =

TLM_UPDATED energy (nJ) =

TLM_COMPLETED energy (nJ) =
```

These energy costs could be assigned to 150 nJ, 170 nJ and 130 nJ respectively.
At this point, all the TLM functions selected for m_initiator_1_phase have been linked to a power model. As final part of the configuration process, since the specific setting has been enabled, the user is asked to define the data concerning the dissipation in idle states:

```
              POWER DISSIPATION IN IDLE STATES

static power (mW) =
```

First of all, the static power shall be specified; this quantity might be set to 1000 mW. Then, the average execution times are required for all the TLM functions selected, in order to evaluate the overall idle time in compliance with the equations (2) and (3) in section 3.2 .

```
Average execution times for the TLM functions selected

         Implemented TLM functions

function: nb_transport_bw
execution time (ns) =


         socket-related TLM functions

function: b_transport
execution time (ns) =

function: nb_transport_fw
execution time (ns) =
```

The execution times are required firstly for the implemented TLM functions; all the times must be reported in nanoseconds. We could assume 10 ns, 5 ns and 7 ns for nb_transport_bw, b_transport and nb_transport_fw respectively.
After that, the specification procedure takes into consideration the last power_module, instanced for m_initiator_2 module:

```
--------------------------------------------
    POWER_MODULE: top.m_initiator_2.m_initiator_1_phase
--------------------------------------------

OPTIONS FOR SPECIFYING THE POWER MODELS

1: interaction with window
2: reading from configuration file
3: no monitoring

select an option (1, 2, or 3) =
```

In this example, such power_module could be excluded from PKtool/TLM analysis by choosing the option 3.
At this point, the configuration process is definitively completed. The SystemC/PKtool simulation resumes its dynamic course with the appearance of an ordinary SystemC simulation. At the end of the simulation, the estimation results for each power_module will reported in distinct text files.

## 8 CONFIGURATION FILE

The window interaction previously described entails the automatic creation of a configuration file for each power_module. Such file reports the configuration data in compliance with a suitable format. The purpose and use of a configuration file are widely explained in the user manual (section 7.3).
In the considered example, an initiator and a target module have been selected for PKtool/TLM analysis; their configuration files are respectively called pk_top.m_initiator_1.m_initiator_1_phase_cfg and pk_top.m_at_target_2_phase_1_cfg.
Taking as reference pk_top.m_initiator_1.m_initiator_1_phase_cfg, the content of a configuration file is represented by the following text:

```
Configuration file    power_module: top.m_initiator_1.m_initiator_1_phase

monitored power_module (Y/N)= Y

enable window menu (Y/N)= Y



      POWER MODEL SPECIFICATION FOR TLM FUNCTIONS


 available TLM power models: 5
 related indexes: 51 52 53 54 55


               IMPLEMENTED FUNCTIONS

 TLM functions selected: nb_transport_bw

 function: nb_transport_bw
 power model: TLM_3  numeric index: 53
 BEGIN_REQ energy (nJ) = 0
 END_REQ energy (nJ) = 200
 BEGIN_RESP energy (nJ) = 250
 END_RESP energy (nJ) = 0


          SOCKET-RELATED FUNCTIONS


 SOCKET: initiator_socket
 TLM functions selected: b_transport  nb_transport_fw

 function: b_transport
 power model: TLM_1  numeric index: 51
 energy (nJ) = 120

 function: nb_transport_fw
 power model: TLM_4  numeric index: 54
```

```
40)   TLM_ACCEPTED energy (nJ) = 150
41)   TLM_UPDATED energy (nJ) = 170
42)   TLM_COMPLETED energy (nJ) = 130
43)
44)
45)       POWER DISSIPATION IN IDLE STATES
46)
47)    static power(mW) = 1000
48)
49)
50)    Execution times of the selected TLM functions
51)
52)
53)    Implemented functions
54)
55)    function: nb_transport_bw
56)    execution time (ns) = 10
57)
58)    Socket: initiator_socket
59)
60)    function: b_transport
61)    execution time (ns) = 5
62)
63)    function: nb_transport_fw
64)    execution time (ns) = 7
```

In the real file there are no line indexes, here reported only for a better reference to the text. At the beginning of the file (lines 3 and 5) there are two settings already discussed in the user manual. Lines 12-13 report some information about the available transaction level power models. Then, the file shows the power model specification for the TLM functions selected for PKtool analysis (lines 16-42). The functions are classified distinguishing the role of the module as calle (implemented functions) and caller (socket-related functions). The second category includes the function called via augmented sockets and defined in external modules. Finally, lines 45-64 specify the data concerning the dissipation in idle states.
This configuration file includes all the possible contributions involved in energy estimations: implemented functions, socket-related functions and idle-state dissipation. More in general, it is possible that a configuration file does not comprise one or two of these contributions, on the basis of how the power_module is configured for PKtool analysis. As an example, the configuration file for the target module (m_at_target_2_phase_1) does not report data for socket-related functions, since such functions have not been selected in the configuration process of this module.
The configuration file is used together with the option 2 of the preliminary window menu to allow an easy and fast power model specification. This represents the best solution when several PKtool simulations have to be carried out with the same configuration data or also with limited modifications. More precisely, the configuration file retains its validity for variations in the data required by the power models or in the execution times of TLM functions. In this case, such modifications can be directly made on the values reported in the file. In the following simulations, the new configuration data will be read from the file by selecting the reading option 2.
The situation is different for variations regarding the selected TLM functions or the power models to be applied, because such modifications cannot be easily specified on a pre-existent configuration file. In this other case, the best approach is to start a PKtool simulation and specify the new

configuration via the window interaction. In this way, PKtool will create automatically an updated configuration file compliant with the new power model specification.

## 9 ANALYSIS RESULTS

At the end of a PKtool/TLM simulation, the analysis results are reported in distinct text files (result files), one for each power_module instanced. The main features of a result file are described in the user manual (section 9). This section is mainly focused on the specific contents due to a PKtool/TLM simulation. For this purpose, we will examine the result file associated to the initiator power_module; the name of this file is pk_top.m_initiator_1.m_initiator_1_phase_res.
In accordance with the configuration data assumed for this power_module, a plausible layout for this file is the following:

```
1)    *************** SIMULATION RESULTS ***************
2)
3)
4)    overall simulation period: [0 - 3800 ns]
5)
6)       POWER/ENERGY DISSIPATION OF IMPLEMENTED TLM FUNCTIONS
7)
8)    function: nb_transport_bw
9)    function calls = 128
10)
11)   energy estimation of BEGIN_REQ phase = 0 J
12)   energy estimation of END_REQ phase = 1.28e-005 J
13)   energy estimation of BEGIN_RESP phase = 1.6e-005 J
14)   energy estimation of END_RESP phase = 0 J
15)
16)   energy estimation = 2.88e-005 J
17)   average power estimation = 7.57895 W
18)
19)
20)      POWER/ENERGY DISSIPATION OF SOCKET TLM FUNCTIONS
21)
22)
23)   SOCKET: initiator_socket
24)
25)   function: b_transport
26)   function calls = 0
27)   energy estimation = 0 J
28)   average power estimation = 0 W
29)
30)   function: nb_transport_fw
31)   function calls = 128
32)
33)   energy estimation of TLM_ACCEPTED status = 9.6e-006 J
34)   energy estimation of TLM_UPDATED status = 0 J
35)   energy estimation of TLM_COMPLETED status = 8.32e-006 J
36)
37)   energy estimation = 1.792e-005 J
38)   average power estimation = 4.71579 W
39)
40)
41)   overall transaction energy estimation: 4.672e-005 J
```

```
42)  average transaction power estimation: 12.2947 W
43)
44)
45)      ENERGY DISSIPATION IN IDLE STATES
46)
47)  overall idle time: 1.624e-006
48)  energy dissipation: 1.624e-006J
49)
50)
51)  OVERALL ENERGY ESTIMATION: 4.8344e-005 J
52)  AVERAGE POWER ESTIMATION:  12.7221 W
```

For each TLM function selected, the file reports the number of calls, the energy estimation and the average power dissipation. The energy estimation is achieved by summing up the energy contributions due to the single function calls and achieved by applying the associated power model. The estimation results are classified distinguishing between the TLM functions implemented by the module (lines 6-17) and the ones called via augmented sockets (lines 20-42). The function b_transport is never called by initiator_socket during a simulation (lines 25-28), for this reason its energy contribution is null. Lines 41-42 specify the overall energy/power estimation due to the transactions occurred during the simulation, obtained by summing up the energy contributions of all the TLM functions. Lines 47-48 report the overall idle time and the energy estimation in idle states. Finally, it is reported the overall power/energy estimation (lines 51-52), achieved from the sum of the overall transaction and idle estimations.

## 10 AVAILABLE POWER MODELS

At the present time, PKtool/TLM provides five power models specific for transaction level power analysis on SystemC/TLM descriptions. Their application is based on the association to a TLM function: the estimations returned by these power models represent the energy dissipation due to single function calls. To this end, some power model defines the energy costs that may concern a function call considering also specific function information. More precisely, these information consist in the characteristic data of a TLM function, i.e. the input parameters and return value (TLM data). The run-time evolution of TLM data is automatically monitored by PKtool/TLM and their current values can be used to determine the energy costs selected as output estimations. The energy costs constituting a power model are treated as static data, and must be declared in the specification procedure at the beginning of a simulation; examples of this step have been shown in Section 7. The available power models are incorporated into the default energy library, in the index range [51-55].

1) TLM_1: this power model is associated to the integer index 51, and estimates the energy dissipation of a function call (Energy) according to the formula:

$$\text{Energy} = E$$

The model returns a fixed energy cost as estimation. The required data are the energy cost (E), which is expressed in nanojoules.

2) TLM_2: this power model is associated to the integer index 52, and estimates the energy dissipation of a function call according to the formula:

$$\text{Energy} = E_{op}$$

where $E_{op}$ is an energy cost depending on the particular operation. More precisely, TLM_2 distinguishes the operation type of a transaction considering the command attribute of generic payload. This attribute can assume three values: TLM_READ_COMMAND, TLM_WRITE_COMMAND and TLM_IGNORE_COMMAND. This means that TLM_2 requires the specification of an energy cost for each of these values.
When executing a TLM function associated to this model, the output estimation is the energy cost of the command specified in the transaction object transported by the function. This power model is applicable only for TLM functions having generic payload as transaction type.

3) TLM_3: this power model is associated to the integer index 53, and estimates the energy dissipation of a function call according to the formula:

$$\text{Energy} = E_{ph}$$

where $E_{ph}$ is an energy cost depending on the particular transaction phase. TLM_3 distinguishes the phase of a transaction as defined by the tlm_phase input parameter. This parameter can assume four values: BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP. This means that TLM_3 requires the specification of an energy cost for each of these values.

When executing a TLM function associated to this model, the output estimation is the energy cost assigned to the phase specified by the function. From the callee and caller side, it is considered the phase value respectively at the beginning and end of the function execution. This power model is applicable only for TLM functions having tlm_phase as input parameter, i.e. nb_transport_fw and nb_transport_bw.

4) TLM_4: this power model is associated to the integer index 54, and estimates the energy dissipation of a function call according to the formula:

$$\text{Energy} = E_{\text{status}}$$

where $E_{\text{status}}$ is an energy cost depending on the particular transaction status specified by the return value of the function. TLM_4 distinguishes the status of a transaction considering the tlm_sync_enum term returned by a TLM function. tlm_sync_enum may assume three values: TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED. This means that TLM_4 requires the specification of an energy cost for each of these values.
When executing a TLM function associated to this model, the output estimation is the energy cost assigned to the tlm_sync_enum term returned by the function. This power model is applicable only for TLM functions having tlm_sync_enum as return type, i.e. nb_transport_fw and nb_transport_bw.

5) TLM_5: this power model is associated to the integer index 55, and estimates the energy dissipation of a function call according to the formula:

$$\text{Energy} = E_{\text{phase-status}}$$

where $E_{\text{phase-status}}$ is an energy cost depending jointly on the transaction phase and status. TLM_5 distinguishes the phase and status of a transaction considering the tlm_phase parameter and tlm_sync_enum return value of a TLM function. In other words, TLM_5 merges the features of TLM_3 and TLM_4 power models, with the capabilities to provide more accurate estimations. In this case twelve possible values should be considered, given by all the distinct (phase, status) couples. This means that TLM_5 requires the specification of an energy cost for each of these combinations.
When executing a TLM function associated to this model, the output estimation is the energy cost of the (phase, status) couple matching the phase and the return status of the. This power model is applicable only for TLM functions having tlm_sync_enum as return type and tlm_phase as input parameter, i.e. nb_transport_fw and nb_transport_bw.