TERMA

TEMU
Device Modelling Guide

Mattias Holm

Version 1.1, 2016-03-01

Table of Contents

1. Introduction

This document provides an overview of how to write models in TEMU. Device models in TEMU are plugins, which can be written in C or C++ (using the C API).

To write a model, it is important to understand the TEMU object system. The object system provides a way to register a *classes*, *properties* and *interfaces*; and a way to instantiate classes to objects.

For users of the System-C, SMP2 or other simulation modelling frameworks, you will find that there are likely several similarities in the TEMU object system to these standards and framework. However, the emulator is intended to work with any external modelling standard and with any language, and the emulator therefor provides its own C level API. It is easy to integrate external modelling standards and frameworks using the external class feature in TEMU.

The TEMU object system is driven by the use of interfaces and interface references (see Section 2.3).

A very important interface for device models is the `MemAccessIface`. This Interface is the interface between the emulator core and the memory system. The interface provides a standardised way to implement read, write and fetch handlers for different memory mapped devices.

The most important interfaces are listed in Section 6

1.1. Coding Conventions

TEMU provides a set of headers that define the API. These headers are located in the include/temu-c/ directory that can be found in the installation directory (normally /opt/temu/x.y.z/). The headers provide a C-API for maximum compatibility (headers are C++ compatible). The headers follow a number of coding conventions.

- The namespace for TEMU is temu, and public functions, types and macros are prefixed with temu_ or TEMU_.
- Enum members are prefixed with teXYZ_ where XYZ is a per enum type abbreviation.
- Interface types (structs containing function pointers) are suffixed with Iface, this suffix is assumed by some macros, so custom interfaces should follow the same suffix rule.
- Interface reference types are suffixed with IfaceRef. This is assumed by some macros, and in-fact, there is a macro TEMU_IFACE_REFERENCE_TYPE that lets you define such reference types.

While the headers have a specific coding style, the only rule that the user have to consider is that custom interfaces must be suffixed with Iface and interface reference types with IfaceRef, although the latter is automatic if the macro is used.

2. The TEMU Object System

Object System Glossary

class

A description of a POD type with properties, interfaces and a name. Realised as a registered struct type.

property

A variable in a POD type, associated with an count (e.g. 1 for scalars, > 1 for arrays), an offset, a type and a name. A property can also have a set of accessor functions registered which enables the association of semantics with a property write or read.

interface

A record of function pointer that is associated with a class. The function pointers take as first argument a pointer to the object.

object

An instance of a class.

As mentioned, TEMU device models are written using the object system. This entail a certain amount of boiler plate code that must be written to register a class in the object system. This boiler plate is typically written in a function with the signature `extern "C" void temu_pluginInit(void)` (or using the macro TEMU_PLUGIN_INIT). This function is called when a TEMU device plugin model is loaded by the object system. Normally, each plugin provides only one device class, but there is nothing preventing a plugin from registering multiple device model classes if needed.

Interfaces are one of the key concepts in the TEMU object system. They provide a way to have standardised functionality associated multiple classes. For example, the `IrqCtrlIface` provides a standard way to raise and lower interrupts. An object implementing interrupt control can use this to provide the functionality in a uniform way. Typical uses of that particular interface are processor models and external interrupt controllers.

2.1. Plugin Mechanism

TEMU is structured around plugins. Plugins are dynamically loaded code libraries that at load time registers any classes that the plugin provides. Plugins can use the TEMU_PLUGIN_INIT macro to define a function that will be called at plugin load time. Loading a plugin can be done using the temu_loadPlugin() function or in the command line interface using the import command. The temu_loadPlugin() function use the normal system paths (i.e. RPATH, LD_LIBRARY_PATH, RUNPATH, and system loader paths), while the command line interface has an additional plugin path variable that can be controlled using the plugin-append-path, plugin-remove-path and plugin-show-paths commands.

To add the plugin initialiser function to your file add the following code to your device model plugin:

Listing 1. Plugin Initialisation Function

```
#include "temu-c/Support/Objsys.h"

TEMU_PLUGIN_INIT
{
  temu_Class *Cls = temu_registerClass(...);
  //...
}
```

The plugin does not have to link to the emulator library since the plugin is loaded by the library. The system linker will resolve any references from the plugin to functions in the emulator libraries.

2.2. Classes and Properties

A class description in TEMU is a registration of a POD-type in the object system, where the fields of the type are also registered as properties. An instance of a class is known as an object.

There are two types of classes, internal and external classes. Internal classes are classes written directly for TEMU, the internal classes are typically associated with properties. All the object system functionality will work on internal classes. When an internal class is instantiated this is done using the object system's functions for creating objects.

External classes are classes that do not use the object system for creation and disposal, but need to integrate with the object system in some ways such as for example expose interfaces to the object system. External classes exists to support the integration of existing device models, by for example wrapping their MMIO handling functionality using the TEMU MemAccessIface.

2.2.1. Internal Classes and Instantiation

As mentioned, internal classes are managed by the TEMU object system. The classes are created by registering a name together with a create and dispose function. These functions are used to allocate and delete memory of objects of the relevant class. In the most basic implementation the create function simply returns a block of data allocated with malloc() or new, and the dispose function pass the object to the counterpart (free() and delete respectively).

All internal classes must inherit temu_Object. This is accomplished by ensuring that the first field in the struct corresponding to the class is temu_Object Super;.

The following snippet shows how to register a new internal class:

Listing 2. Registering an Internal Class

```
typedef struct {
  temu_Object Super; // Must be first field in struct
  // ...
} MyDevice;

// Register new internal class
temu_Class*
temu_registerClass(const char *ClsName,
                   temu_ObjectCreateFunc Create,
                   temu_ObjectDisposeFunc Dispose);

// Creating objects of internal classes
void* temu_createObject(const char *ClsName, const char *ObjName,
                        const temu_CreateArg *Args);
```

2.2.2. External Classes and Object Registration

External classes are managed outside TEMU, they therefor only have a class name and associated interfaces (and possibly also properties). Creation and destruction of external objects is done outside the TEMU runtime, as such, objects of external types, must be registered manually with the TEMU runtime, while at the same time specifying which class the object belong to.

Listing 3. Registering an External Class

```
temu_Class* temu_registerExternalClass(const char *ClsName);

// Use this to add an interface to any class
void
temu_addInterface(
  temu_Class *Cls,

  // Name of interface (used in connect calls)
  const char *IfaceName,

  // Type of interface (reserved),
  // should be same as C-type name minus namespace prefixes
  const char *IfaceType,
  void *Iface, // Pointer to interface
  int Count, // Reserved for future use (should be 1)
  const char *Doc) // Documentation string for interface

// To add objects of external class use the addObject function
void* temu_addObject(const char *ClsName,
                     const char *ObjName,
             void *Obj);
```

2.3. Interfaces and Interface References

An interface is a collection of function pointers, normally stored in a struct. These interface types are used to allow for model compatibility with different subsystems. An interface is registered to a class at class creation time. The first parameter of each function in an interface should by convention take an object pointer (i.e. a self or this pointer of type void*), as the object system API is expressed in C, the self/this pointer must be passed explicitly.

Interface references however are properties that refer to an object and interface pair, that is the interface reference bundle the self/this pointer together with the interface pointer. This is realised as a struct with two pointers (object and interface pointer, the fields are named Obj and Iface respectively). The object system header (temu-c/Support/Objsys.h) provides a macro to define a typed version of the interface reference struct. This macro is named TEMU_IFACE_REFERENCE_TYPE and will simply define a struct with the suffix IfaceRef. Note that the interface type (i.e. the struct with the function pointers) must have a suffix Iface for this macro to work.

There is also an untyped interface reference type with the name temu_IfaceRef.

Interface references are central to understand in order to be able to work with the object graph provided by the object system.

The emulator provides a function with the name temu_connect(), this function can be used to connect an interface reference in a specific object to an object and an interface implemented the latter object's class.

The following code snippet illustrate how to connect objects using the TEMU API.

Listing 4. Connecting Objects

```
// Equivalent to CLI: connect a=obja.propName b=objb:SomeIface
temu_connect(ObjA, "propName", ObjB, "SomeIface");
```

The first parameter to temu_connect() is an object pointer, the second parameter is the property name (which must be either an interface reference property or an interface reference array property). Third is the destination object pointer, and fourth is the interface name for the destination object. Note that it is legal to connect an a property to an interface implemented by the object itself, to do this, pass the same pointer to the source and destination object arguments.

The connection function essentially fills the named property with a pointer for the destination object and the pointer for the interface struct, which is looked up by name. To call some function in the connected object one simply call the function in the interface, passing the object pointer as first parameter as illustrated in the following snippet:

Listing 5. Calling a Function in an Interface

```
ObjA->PropName.Iface->someFunc(ObjA->PropName.Obj);
```

It is possible to connect interface reference properties in two ways. In the first case the property is of type teTY_IfaceRef, in that case the connection is done by looking for the first free entry in the property field. In case all entries in the property is already connected, the connection operation fails.

The second case is when the type is teTY_IfaceRefArray, which is a dynamic array of interface references. This connection will always succeed assuming the destination object and interface are valid (and unless the system runs out of memory).

Temu_connect() returns non-zero on failure (i.e. invalid property name, invalid interface name, invalid object pointers).

Connecting an interface reference property which is a static array will place the interface reference in the first free array entry (i.e. where the object and interface pointer pair are both NULL).

2.4. The Object Graph

The object system provides a registry of objects. The objects are connected to each other using interface references. This object connectivity is known as the object graph. As mentioned in Section 2.3, the objects are connected using the temu_connect() function.

3. Simple Operations

This section describes how to accomplish the most common tasks when writing models for TEMU.

3.1. Memory Mapped I/O and Registers

The primary ways to get an MMIO transaction is to implement the MemAccessIface in your model. The MemAccessIface has three functions (fetch, read and write). As arguments the functions take the object (device model pointer) and a MemTransaction object. The transaction object is important to fully understand.

By implementing the MemAccessIface the device model becomes compatible with the memory mapping functions.

The following snippet illustrates how to add support for the memory access interface:

```
#include "temu-c/Support/Objsys.h"
#include "temu-c/Memory/Memory.h"

void readFunc(void *Obj, temu_MemTransaction *MT);
void writeFunc(void *Obj, temu_MemTransaction *MT);

temu_MemAccessIface MemAccessIface = {
  NULL, // fetchFunc is optional (only used for RAM and ROM models)
  readFunc,
  writeFunc,
};

TEMU_PLUGIN_INIT
{
  temu_Class *Cls = temu_registerClass("MyClass", create, dispose);
  temu_addInterface(Cls, "MemAccessIface", "MemAccessIface",
                    &MemAccessIface);
}
```

The memory transaction object has an Initiator pointer, if the model needs to call functions that exits the emulator core directly (via longjump, the relevant functions in the CPU interface are tagged as noreturn functions), this should ONLY be done if there is an initiator object specified. Without an initiator object, the transaction is initiated from elsewhere (e.g. from a manual MMU table walk).

MODELS SHOULD NOT USE FUNCTIONS THAT LONGJMP TO THE EMULATOR CORE IF THERE IS NO INITIATOR OBJECT.

An object implementing the MemAccessIface can be mapped into a memory space. Normally the memory-mapped I/O model will provide a set of registers. There is no special register type, but most registers are implemented as properties, where the read and write functions act as register read and writes. It is necessary for the device model to provide functions implementing the MemAccessIface that dispatch reads and writes to the correct functions.

The normal way to do the dispatching is via a switch on the Offset field in the memory transaction object.

Listing 6. Memory Mapped I/O Interface Usage

```
#include "temu-c/Memory/Memory.h"

void
readFunc(void *Obj, temu_MemTransaction *MT)
{
  switch (Mt->Offset) {
  case 0:
    // Unwrap property
    Mt->Value = temu_propValueU32(readRegA(Obj, 0));
    break;
  case 4:
    Mt->Value = temu_propValueU32(readRegB(Obj, 0));
    break;
  }
  Mt->Cycles = 0; // Cost for this memory transaction in cycles
}
```

Registers are properties in the structure which have read and write handlers implementing the semantics. The properties can be written to from the API and the command line interface. This way unit testing of device models is very simple since device models can be stimulated without writing custom code for the target processor.

3.2. Device Resets and Mappings

The DeviceIface can be implemented by a device model for handling resets and mapping of devices to memory (a device should normally not need to know where it is mapped, but it may be used for automatic update of plug-and-play info that needs to reflect the MMIO address mappings).

The DeviceIface must be implemented in case the device must support resets. The reset function takes an int as parameter, specifying the reset type. By convention, 0 means a cold reset, while 1 means a warm reset.

```
#include "temu-c/Models/Device.h"

static void
reset(void *Obj, int ResetType)
{
  if (!ResetType) {
    // Cold reset
  }
  //...
}

static void
mapDevice(void *Obj, uint64_t Address, uint64_t Len)
{
  temu_logInfo(Obj, "device was mapped at %0.8x", (uint32_t)Address);
}

temu_DeviceIface DeviceIface = {
  reset,
  mapDevice,
};

TEMU_PLUGIN_INIT
{
  temu_Class *Cls = ...
  // To register device interface call the following in the plugin:
  // init function:
  temu_addInterface(Cls, "DeviceIface", "DeviceIface", &DeviceIface,
                    1, NULL);
}
```

3.3. Posting Timed Events

A device model that need to post timed events need to have an event queue object associated to itself. The event queue is provided by the CPU objects. That means that there are one event queue per CPU. For the cases where an event must be invoked at a synchronised time stamp (i.e. all CPUs having reached a specific time), then events can be posted by setting the teSE_Machine sync qualifier when posting the events.

Events are registered at object construction time and have an associated ID. There are a number of ways to register events. This is done with the functions:

- temu_eventPublishStruct()
- temu_eventPublish()

The struct publication allows you to embed an event structure inside your object or to subclass the event struct (i.e. using the temu_Event struct type as the first field (normally called Super)). The

second function creates an event struct in the global event struct registry, both functions return the Event ID which is globally unique in your program. This event ID is then typically saved in your class as a separate field (this field should not be registered as a property, event IDs are not guaranteed to remain the same the next run).

In order to post events, there are four functions available:

- temu_eventPostCycles()
- temu_eventPostNanos()
- temu_eventPostSecs()
- temu_eventPostStack()

Each of these take as parameter a queue object (a CPU object) and the event ID that was returned by thee temu_ventPublishXXX() functions.

All functions except the stack posting one post events relative to the current time as understood by the queue object.

The last argument to the functions are the "Sync" parameter. At present this can be either teSE_Cpu, or teSE_Machine, this flag indicates whether the event should go on the CPU queue, or if it should be forwarded to the machine object. If the event is forwarded, it is:

1. The delta is converted to a nanosecond offset.
2. Rounded to the start of the next machine time quanta if less than it.
3. Inserted in the relevant queue.

Stacked events on the other hand, will be executed after the current instruction. A stacked event that is machine synchronised will be executed at the end of the current time quanta.

```
// Header for event interface
#include "temu-c/Support/Event.h"

// Stack event (will be invoked after this instruction
// (or event) is handled)
temu_eventPostStack(Dev->Super.TimeSource, Dev->MyEventID, teSE_Cpu);

// Post an event 123 cycles in the future
temu_eventPostCycles(Dev->Super.TimeSource, Dev->MyEventID, 123, teSE_Cpu);

// Post a synchronised event at 42 ns in the future
temu_eventPostNanos(Dev->Super.TimeSource, Dev->MyEventID, 42, teSE_Machine);

// Get delta time in cycles (useful in timer models)
delta = temu_eventGetCycles(Dev->Super.TimeSource, Dev->MyEventID);

// Deschedule event identified by function and sender pair
temu_eventDeschedule(Dev->Super.TimeSource, Dev->MyEventID);

// The object create function which is responsible for creating the
// object is responsible for registering the events.

void*
create(const char *Name, int Argc, const temu_CreateArg *Argv)
{
  MyDevice *Dev = new MyDevice;
  Dev->MyEventID = temu_eventPublish("myEventName", Dev, MyEventFunc);
}
```

3.3.1. Posting Initial Events

When an object is created, it does not have an event queue associated with it, thus it is not possible to post events (unless you set the event queue manually in your constructor). To overcome this issue it is possible to implement the ObjectIface and its member timeSourceSet. This will be called after temu_setTimeSource() has set the time source property. When this function is called on your device model, it is safe to post events.

3.4. Raising Interrupts

It is common that device models need to raise an interrupt. For this purpose there are two interfaces to consider. The IrqCtrlIface which is used to raise and / or lower interrupts. The IrqCtrlIface is implemented by processor models and interrupt controllers.

There is also IrqClientIface which is implemented by interrupt controllers so they support lasy interrupt evaluation. A device model will normally only need to bother about the IrqCtrlIface.

```
// Header for interface
#include "temu-c/Models/IrqController.h"

// Raise IRQ 1
Obj->IrqCtrl.Iface->raiseInterrupt(Obj->IrqCtrl.Obj, 1);

// Lower IRQ 1
Obj->IrqCtrl.Iface->lowerInterrupt(Obj->IrqCtrl.Obj, 1);
```

In most cases, a device model should be connected to an interrupt controller and not directly to the CPU.

When a multi-core system is emulated, an interrupt controller may need to distribute interrupts to different processors. There are the following alternatives for this:

- An interrupt is sent to the same CPU who initiated the interrupt (e.g. by MMIO or event execution).
- An interrupt is sent to another CPU than the one initiating the interrupt (e.g. an Inter-Processor Interrupt (IPI)).
- An interrupt is sent to several CPUs (multi- or broadcast interrupt).

In the case an IRQ is raised on the initating CPU (or from a synchronised event) then the IRQ will be taken in natural order.

In the case the interrupt is raised on another CPU, then the IRQ raising time will be at the start of the current quanta or at the start of the next quanta. That is, the IRQ may be raised earlier in apparent time for the other CPU.

3.5. Accessing Memory Contents

Some devices may need to access memory content in memory models, this can be done using the MemoryIface interface. The interface provides functions to read and write byte blocks from emulated memory. The interface is implemented by memory spaces, but would typically also be provided by memory models, CPU models and IOMMU models.

By convention, a memory space handles physical address offsets, while a CPU model would handle virtual addresses in this interface. It is only allowed to do memory content access in one device at a time. That is if one attempts to read or write a block which will span multiple device mappings (e.g. end of ROM followed by start of RAM), then the memory access will fail.

Memory content is accessed using the MemoryIface. Note that the model should normally be connected to the memory space for this. Although, some systems that contain an IOMMU would connect the device models directly to the IOMMU model instead.

Note that the read and write bytes functions will not trigger side effects. I.e. if the access is intended to be routed to a device model, the memory transaction interface should be used (but setting the

initiator to NULL in the transaction object).

```
// Writing 128 bytes, data is given as an array of 32-bit words
Dev->Mem.Iface->writeBytes(Dev->Mem.Obj, 0x40000000, 128,
                           &Data[0], 2);

// Reading data, out data will be an array of 32-bit words
Dev->Mem.Iface->readBytes(Dev->Mem.Obj, &Data[0], 0x40000000, 128, 2);
```

4. Complex Operations

4.1. Custom Checkpointing

While in most cases, checkpoint-support is automatic for device model simply by registering the properties with the class. In some cases it may make sense to implement custom checkpoints. To implement a custom checkpoint a class must implement the ObjectIface (temu-c/Support/Objsys.h). This interface is optional but allows for custom serialisation routines to be implemented.

When saving a checkpoint, the system first writes out all registered properties in an object (this process is known as serialisation). After the property serialisation, the optional serialise function in the (optional) ObjectIface is called.

When restoring a checkpoint, the system first creates all objects by calling the constructors registered for a class, after this properties are restored, thirdly, if an object is of a class that implements the deserialise function that function is called.

When restoring a checkpoint, the object constructors are called without arguments.

The checkpointing functions take two parameters, BaseName is the name of the file where the checkpoint will be written, and Ctxt is a context pointer that can be used to insert and query additional property values in the checkpoint file.

A typical use for the BaseName parameter is in the RAM and ROM models. These, by default, dump the raw data into binary files which will be named by adding a suffix to BaseName.

To serialise a special property in the custom checkpointing interface, the following functions can be used:

```
// Write out a property
void temu_serialiseProp(void *Ctxt, const char *Name, temu_Type Typ,
                        int Count, void *Data);

// The following can be used to deserialise properties for objects.
// Get length of the property in the given context (i.e. number of
// elements)
int temu_checkpointGetLength(void *Ctxt, const char *Name);

// Get a value for the indexed property in the given context
temu_Propval temu_checkpointGetValue(void *Ctxt, const char *Name, int Idx);
```

To implement custom serialisation the ObjectIface is implemented. The following example shows how to match the serialise and deserialise functions for saving and restoring checkpoints:

```
void
serialise(void *Obj, const char *BaseName, void *Ctxt)
{
  uint32_t Extra = 123;
  temu_serialiseProp(Ctxt, "myExtraProp", teTY_U32, 1, &Extra);
}

void
deserialise(void *Obj, const char *BaseName, void *Ctxt)
{
  assert(temu_checkpointGetLength(Ctxt, "myExtraProp") == 1);
  temu_Propval PV = temu_checkpointGetValue(Ctxt, "myExtraProp", 0);
}

temu_ObjectIface ObjIface = {
  serialise,
  deserialise,
  NULL, // checkSanity
};
```

4.2. DMA Access Emulation

To emulate DMA, the standard approach is as follows:

1. When DMA transaction starts, the whole data block is copied and the estimated time is computed and an event is posted at the end of transaction time.
2. When the transaction event is called, the event handler raises an interrupt.

Transferring all the data when posting the event is preferred over transferring it in the event handler. This way data may be sourced from the stack without problems, if data must be copied to memory at the event time, then a buffer must be allocated on the heap for this purpose.

The following example illustrates emulation of DMA transactions.

Listing 7. DMA Emulation Example

```
void
dmaFinished(void *Sender, void *Data)
{
  MyDevice *Dev = (MyDevice*)Sender;
  Dev->IrqCtrl.Iface->raiseInterrupt(Dev->IrqCtrl.Obj, 1);
}

void
dmaStart(MyDevice *Dev)
{
  uint64_t TransactionTime = MyDevice->DmaSize * NsPerByte;
  Dev->Mem.Iface->writeBytes(Dev->Mem.Obj, Dev->DmaAddr, Dev->DmaSize,
                             Dev->TransactionData, 2);
  Dev->Queue.Iface->postDeltaEvent(Dev->Queue.Obj, dmaFinished, Dev, NULL,
                                   TransactionTime, TEMU_EVENT_NS);
}
```

4.3. Proxy Objects

An interesting debugging technique is to use proxy objects where one implement an interface and simply forwards the calls to the same interface but as provided by another class. This is called a proxy object, and such an object can for example log calls or provide other interesting diagnostics.

Similar techniques can also be used to implement cache models in the emulator.

4.4. Using and Implementing Bus Models

The complexity of issuing a transaction on a non-MMIO bus depends on the type of bus. A point to point bus (e.g. serial) may be implemented by simple interfaces in the models where models connect directly to each other and do not strictly speaking need a separate bus model class.

However, multi-point links typically needs a bus model to work, this bus-model is for example responsible for routing messages to the correct model.

Even point-to-point buses like serial could use a bus model object in order to ensure that a bus client behaves appropriately (e.g. verify bandwidth usage), or to implement hardware flow-control emulation, or to be able to inject errors.

There is no 'generic' bus model as connecting a spacewire connector to a serial port or to a milbus makes no sense. Thus each bus needs its own interface (and optionally a bus-model component). This approach also ensures a degree of type-safety.

When constructing a more complex bus model, one way is to provide a connect function, and ensure that users simply set the bus property in the models connecting to the bus. When this property is set, the connect function in the bus-object is called with whichever parameters are relevant for the bus in question.

There are also other issues to taken into account. For example, if the bus model is frame-based, in order to be able to interrupt a transaction (e.g. after half of it was sent), it may be useful to be able to issue two messages issued of one. That is, one begin transaction and one end transaction message. The exact requirements depend on the accuracy of the model one need. However, if one only models working transactions, then often a single message is needed.

Listing 8. Bus Model Example

```
typedef struct {
  uint16_t Dest;
  uint16_t *Data;
} MyTransaction;

typedef struct {
  void (*receive)(void *Obj, MyTransaction *T)
} MyDeviceIface;
TEMU_IFACE_REFERENCE_TYPE(MyDevice);

typedef struct {
  void (*connect)(void *Bus, uint16_t Addr, MyDeviceIfaceRef Device)
  void (*send)(void *Obj, MyTransaction *T)
} MyBusIface;
TEMU_IFACE_REFERENCE_TYPE(MyBus);

typedef struct {
  MyBusIfaceRef Bus;
} MyDevice;

MyDeviceIface MyIfaceImpl = {
  receive
};

// Connect to the bus when the bus property is written
void
writeBus(void *Obj, temu_Propval PV, int Idx)
{
  MyDevice *Dev = Obj;

  Dev->Bus = temu_propValueIfaceRef(PV);

  MyDeviceIfaceRef DevIface = {
    Dev, MyIfaceImpl;
  };

  Dev->Bus.Iface->connect(Dev->Bus.Obj, 123, DevIface);
}
```

5. A Simple Model Example

In this example we implement a model with two registers that can be accessed using the read and write procedures in the memory access interface. We have a device which contain two registers (*RegA* and *RegB*). These registers are mapped at offset 0 and 4 from the device base address.

To implement this model, we implement read and write functions for registers a and b, memory transaction handlers which decode the offset and calls the relevant read or write handler. The read and write functions are important as they provide a standard interface for the TEMU runtime to call register reads and writes; this is especially useful in device model tests which does not need to deal with the rather complex memory access interface. In addition, it simplifies device debugging for the same reason when loading a device in the TEMU command line interface.

The model here can be compiled with GCC or clang using:

```
# With GCC
g++ -fPIC -shared mydevice.cpp -o libMyDevice.so

# With clang:
clang++ -fPIC -shared mydevice.cpp -o libMyDevice.so
```

Note that you do not need to link to the support library (libTEMUSupport.so) as the symbols from that library are resolved when the plugin is loaded.

The built model can be loaded form the command line with the import command.

Listing 9. Example Class

```
#include "temu-c/Support/Objsys.h"
#include "temu-c/Memory/Memory.h"

#include <stdint.h>

typedef struct {
  temu_Object Super;

  uint32_t RegA;
  uint32_t RegB;
} MyDevice;

void*
create(const char *Name, int Argc, const temu_CreateArg *Argv)
{
  MyDevice *Device = new MyDevice;
  memset(Device, 0, sizeof(MyDevice));

  return Device;
}

void
destroy(void *Obj)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  delete Dev;
```

```
}

void
writeRegA(void *Obj, temu_Propval Val, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register write goes in here

  Dev->RegA = temu_propValueU32(Val);
}


temu_Propval
readRegA(void *Obj, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register read goes in here

  return temu_makePropU32(Dev->RegA);
}


void
writeRegB(void *Obj, temu_Propval Val, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register write goes in here

  Dev->RegB = temu_propValueU32(Val);
}

temu_Propval
readRegB(void *Obj, int Idx)
{
  MyDevice *Dev = reinterpret_cast<MyDevice*>(Obj);
  // Semantics of register read goes in here

  return temu_makePropU32(Dev->RegB);
}


void
readFunc(void *Obj, temu_MemTransaction *Mt)
{
  switch (Mt->offset) {
  case 0:
    Mt->Value = temu_propValueU32(readRegA(Obj, 0));
    break;
  case 4:
```

```
    Mt->Value = temu_propValueU32(readRegB(Obj, 0));
    break;
  }
  Mt->Cycles = 0;
}

void
writeFunc(void *Obj, temu_MemTransaction *Mt)
{
  switch (Mt->offset) {
  case 0:
    writeRegA(Obj, temu_makePropU32(Mt->Value), 0);
    break;
  case 4:
    writeRegB(Obj, temu_makePropU32(Mt->Value), 0);
    break;
  }
  Mt->Cycles = 0;
}

temu_MemAccessIface MemAccessIface = {
  NULL, // fetch
  readFunc,
  writeFunc,
};

extern "C" void
temu_pluginInit(void)
{
  temu_Class *Cls = temu_registerClass("MyDevice", create, destroy);

  temu_addProperty(Cls, "regA", teTY_U32, 1,
                   offsetof(MyDevice, RegA),
           writeRegA, readRegA,
           "Register A");

  temu_addProperty(Cls, "regB", teTY_U32, 1,
                   offsetof(MyDevice, RegB),
           writeRegB, readRegB,
           "Register B");

  temu_addInterface(Cls, "MemAccessIface", &MemAccessIface);
}
```

6. Important Interfaces

Listing 10. Memory Access Interface

```
#include "temu-c/Memory/Memory.h"

typedef struct temu_MemTransaction {
  uint64_t Va;      //!< 64 bit virtual for unified 32/64 bit interface.
  uint64_t Pa;      //!< 64 bit physical address
  uint64_t Value;   //!< Resulting value (or written value)

  //! Log size of the transaction size it is at most the size of the
  //! CPUs max bus size. In case of SPARCv8, this is 4 bytes (double
  //! words are issued as two accesses).
  uint8_t Size;

  //! Used for device models, this will be filled in with the offset
  //! from the start address of the device (note it is in practice
  //! possible to add a device at multiple locations (which happens in
  //! some rare cases)).
  uint64_t Offset;
  void *Initiator; //!< Initiator of the transaction
  void *Page;      //!< Page pointer (for caching)
  uint64_t Cycles; //!< Cycle cost for memory access
} temu_MemTransaction;

// Exposed to the emulator core by a memory object.
typedef struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
  void (*read)(void *Obj, temu_MemTransaction *Mt);
  void (*write)(void *Obj, temu_MemTransaction *Mt);
} temu_MemAccessIface;
```

Listing 11. IRQ Interface

```
#include "temu-c/Models/IrqController.h"

typedef struct temu_IrqControllerIface {
  void (*raiseInterrupt)(void *Obj, uint8_t Irq);
  void (*ackInterrupt)(void *Obj, uint8_t Irq);
} temu_IrqCtrlIface;
```

Listing 12. Device Interface

```c
#include "temu-c/Models/Device.h"

typedef struct temu_DeviceIface {
  void (*reset)(void *Obj, int ResetType);
  void (*mapDevice)(void *Obj, uint64_t Address, uint64_t Len);
} temu_DeviceIface;
```