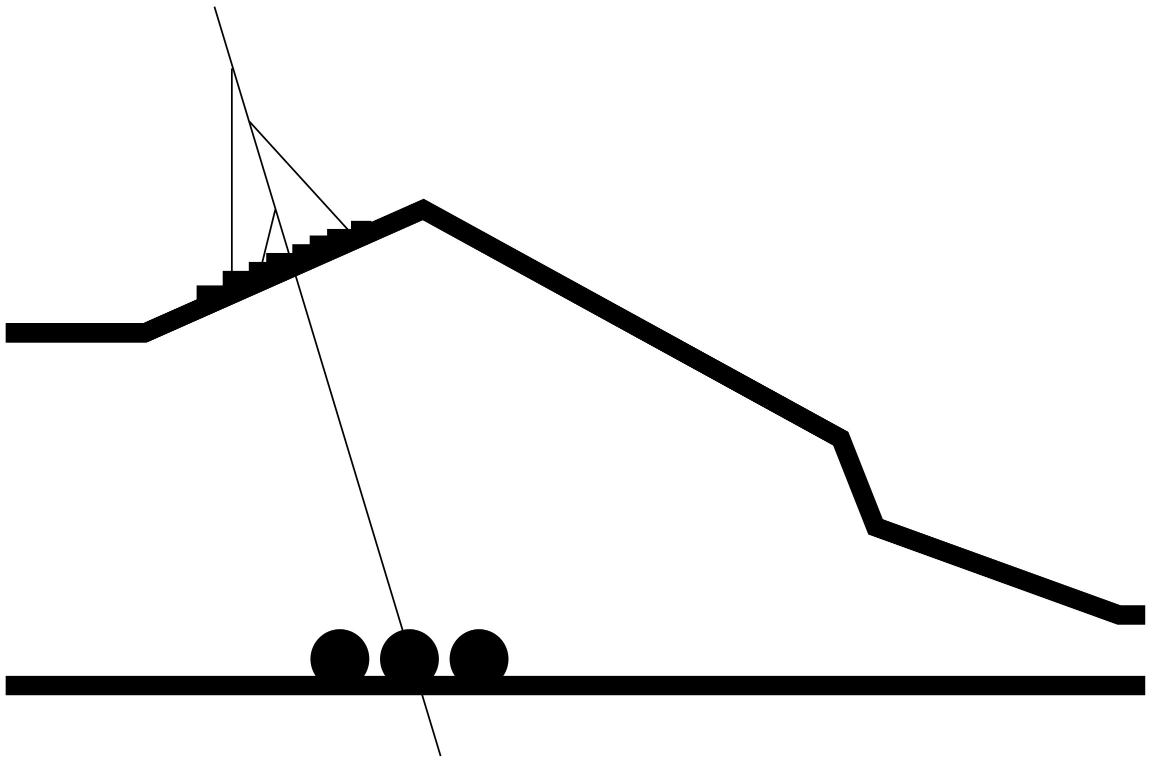
LNGS/TC-02/06
June 2006



# V-DAS (A Versatile Data Acquisition software): The interface to the VME bus and the configuration file interpreter

M. Di Paolo Emilio[a,b], S. Stalio[a],

INFN - Laboratori Nazionali del Gran Sasso

**INFN - Istituto Nazionale di Fisica Nucleare**

**Laboratori Nazionali del Gran Sasso**

**LNGS/TC-02/06**

**June 2006**

# V-DAS (A Versatile Data Acquisition software): The interface to the VME bus and the configuration file interpreter

M. Di Paolo Emilio[a,b], S. Stalio[a],

[a] *INFN, Laboratori Nazionali del Gran Sasso Assergi (AQ) - Italy*

[b] *Dipartimento di Fisica - Universita' degli studi dell'Aquila - Italy*

**Abstract**

A new data acquisition system (DAQ) named V-DAS and based on the VME bus is presented. The system has been developed for the data acquisition system of the gravitational antennas belonging to the ROG group. In this article we will describe the software for the communication with the VME bus and the management of the VME boards by means of a text file (configuration file).

## 1 Introduction

V-DAS is a software written using the C language for the management of VME based DAQ systems. It has been developed at Laboratori Nazionali di Frascati and at Laboratori Nazionali del Gran Sasso. The VME bus (Versa Module Europa) is a flexible open-ended bus system based on the Eurocard standard. It was introduced by Motorola, Phillips, Thompson, and Mostek in 1981. VME bus was intended to be a flexible environment supporting a variety of computing intensive tasks, and is now a very popular protocol in the computer industry. It is defined by the IEEE 1014-1987 standard. The system is modular and follows the Eurocard standard. VME card cages contain 21 slots, the first of which must be used as a crate manager.
The idea that led us to the realization of V-DAS has been the necessity of creating, starting from the vme_universe drivers and libraries for the VME bus and the standard C libraries, a new set of functions and structures that assures the easy management of VME based DAQ systems (figure 1).
V-DAS has been originally developed for the acquisition of data generated by the gravitational antennas (Nautilus and Explorer) belonging to the ROG (www.lnf.infn.it/esperimenti/rog) group. The system architecture relies on a VME crate managed by an Intel-based crate controller running the Linux operating system[1].

## 2 V-DAS architecture

V-DAS is composed of 5 subsystems, each having a specific function:

- VME bus interface: implements the communication with the boards mounted in the VME crate.
- Data writing: takes care of writing acquired data on structured data files.
- Configuration file interpreter: reads and parses the configuration file and sets up the DAQ.
- Error handler: manages errors that may show up during data taking (network problems, VME bus errors, disk access problems, ...)
- Network data transfer manager: takes care of transferring acquired data from the VME crate manager to an optional data storage host via an Ethernet connection.

## 3 The interface to the VME bus

The functions and data structures implemented in this module of the V-DAS software control the initialization of the VME bus, data reading and writing from/on any register

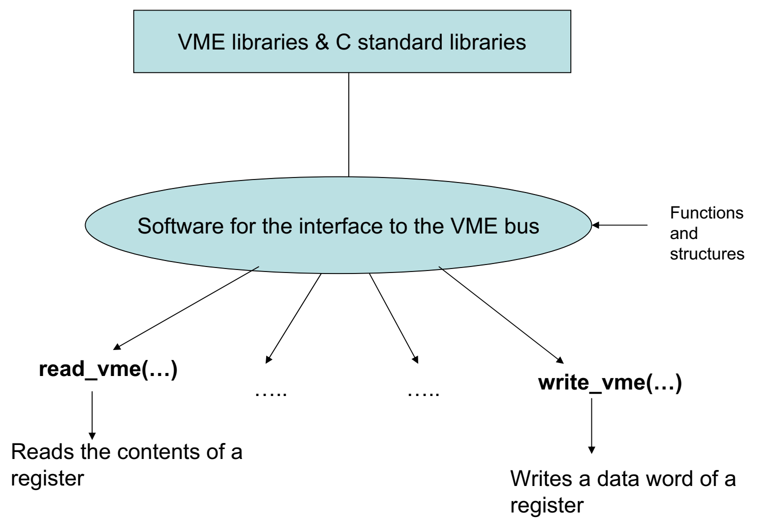

VME libraries & C standard libraries
Software for the interface to the VME bus
Functions and structures
read_vme(…)
…..
…..
write_vme(…)
Reads the contents of a register
Writes a data word of a register

Figure 1: Composition of the software for the interface to the VME bus

or memory area, the execution of multiple operations on different register and other ancillary operations.

## 3.1 The data structure

In V-DAS all the registers that can be accessed on the VME bus are organized in a hierarchic object structure. This structure is made of *components* (the lower level), *equipments* and *triggers*. The *component* object keeps all the information that is needed in order to fully identify a VME bus data register (or memory area). The *equipment* object is a list of *components*. The *trigger* object is a list of equipments.

### 3.1.1 The component object

The *component* object is defined by the following data structure:

```
typedef struct
{
char module[40];
char regname[40];
char label[50];
char dw[20];
char action[20];
char am[20];
unsigned long base;
unsigned long offset;
unsigned long size;
int bus;
uint8_t *value_read8; (for a data width of 1 byte)
uint16_t *value_read16; (for a data width of 2 byte)
uint32_t *value_read32; (for a data width of 4 byte)
uint64_t *value_read64; (for a data width of 8 byte)
unsigned long value_to_write;
unsigned long expected_value;
vme_master_handle_t dma_handle;
vme_dma_handle_t dma_handle;
} vme_component;
```

The value of the *module* variable represents the name of the VME board to which the register belongs. The *regname* variable represents the VME register name, while the *label* variable identifies the component. The *dw* variable identifies the data width. The *am* variable identifies the type of VME bus access (Address Modifier). The *action* variable identifies the type of operation to be performed on the registers, accepted values are:

- read: read data register (or memory area). The content of the register is saved in the *value_read* variable.
- write: write data (found on the *value* field) on the selected register.
- read_verify: read data register and compare the result with the value of the *value_expected* parameter.
- read_loop: repeat readout of the data register until the result is equal to the value of the *value_expected* parameter.

The *handle* variable identifies the VME register bus, while the *dma_handle* variable identifies the VME bus memory area if the component represents a DMA register.
The VME board is identified by a base address (hexadecimal) which is saved in the *base* parameter. Each register is identified by an offset which is the offset of the register address with respect to the base address. This value is saved in the *offset* parameter. Finally the

incremental number of the VME crate to which the crate controller belongs is saved in the *bus* variable.
The *vme_component_list* data structure contains a list of all declared *components*.

```
typedef struct
{ vme_component components[MAX_NUM_COMPONENTS];
int num_elements;
} vme_component_list;
```

The number of declared *components* is saved in the *num_elements* variable. Each register is described in the *components* array of *vme_component* objects.
*MAX_NUM_COMPONENTS* is a constant that defines the maximum number of allowed *components*.

### 3.1.2 The equipment object

The *equipment* object is defined by the *vme_equipment* data structure:

```
typedef struct
{
int element[MAX_NUM_COMPONENTS_IN_EQUIPMENT];
int size;
char name[120];
time_t time;
}vme_equipment;
```

The number of *components* of an *equipment* is saved in the *size* variable. The *name* variable is an arbitrary string. The list of *components* contained in an *equipment* is kept in the *element* array. The *time* variable is an integer that controls the execution period of each *equipment*. If *time* is -1 the *equipment* is executed only once at the beginning of data taking. If *time* is 0 the equipment is executed whenever possible. If *time* has a positive value, this value represents the period (in seconds) of the *equipment* execution.
*MAX_NUM_COMPONENTS_IN_EQUIPMENTS* is a constant. It defines the maximum number of *components* that an *equipment* can contain.

The *vme_equipment_list* data structure contains a list of all declared *equipments*.

```
typedef struct
{
vme_equipment equipment[MAX_NUM_EQUIPMENT];
int num_elements;
}vme_equipment_list;
```

The number of *equipments* is saved in the *num_elements* variable. All *equipments* are described in the *equipment* array of *vme_equiment* objects.
*MAX_NUM_EQUIPMENT* is a constant that defines the maximum number of *equipments*.

### 3.1.3 The trigger object

The *vme_trigger* structure contains a full description of a *trigger*. The number of *equipments* belonging to a *trigger* is saved in the *size* variable. The *name* variable is an arbitrary string. The list of *equipments* contained in the *trigger* is kept in the *equip_list* array. The *time* variable is an integer that controls the execution period of each *trigger*. If *time* is -1 the *trigger* is executed only once at the beginning of data taking. If *time* is 0 the *trigger* is executed whenever possible. If *time* has a positive value, this value represents the period (in seconds) of the *trigger* execution. *MAX_NUM_EQUIPMENTS* is a constant that defines the maximum number of *equipments* that a *trigger* can contain.

```
typedef struct
{
char name[120];
int size;
vme_equipment equip_list[MAX_NUM_EQUIPMENT];
time_t time;
}vme_trigger;
```

The *vme_trigger_list* data structure is a list of all declared *triggers*.

```
typedef struct
{
vme_trigger trigger[MAX_NUM_TRIGGER];
int num_elements;
}vme_trigger_list;
```

The number of *triggers* is saved in the *num_elements* variable. Each *trigger* is described in the *trigger* array of *vme_trigger* objects. *MAX_NUM_TRIGGER* is a constant that defines the maximum accepted number of *triggers*.

## 3.2 Basic functions

We will describe in detail the most important V-DAS functions that permit reading and writing on VME bus registers.

**vme_bus_handle_t init_bus_vme(vme_bus_handle_t handle)**

The *init_bus_vme()* function initializes the VME bus. It creates a pointer (*handle*) that identifies the bus. The function returns the pointer to the initialized bus.
After initializing the VME bus, a "window" must be created for each *component*:

**vme_component create_vme_window(vme_bus_handle_t handle,
vme_component reg**)

The *create_vme_window()* function creates a "window" for the *reg* object of type *vme_component*. The "window" is a software interface that permits access to the *reg* data register. The bus is identified by the *handle* variable. The function returns a *vme_component* object.
This operation is fundamental and it must be performed before accessing any register.

The VME "window" needs then to be mapped to a memory location on the acquisition computer. This operation is performed by the *map_vme_window()* function.

**vme_component map_vme_window(vme_bus_handle_t handle,
vme_component reg**

This function can only map single word registers (not DMA memory areas) and returns a *vme_component* object. The *map_vme_window()* function is usually run only one time for each *component* during data acquisition.

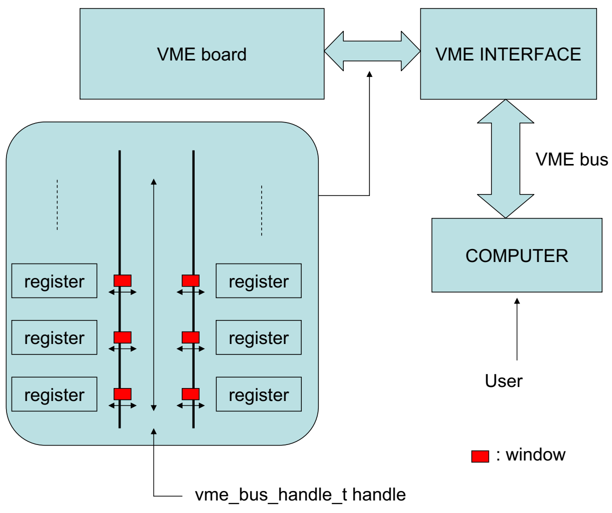


Figure 2: VME window

After running the *create_vme_window()* function and the *map_vme_window()* function on a register, this register is ready for reading or writing.

## 3.3 Functions for reading and writing on registers

The *read_vme()* function reads data from a VME register. All access parameters are stored in the *reg* object. Data read is also stored in the *reg* object. Accepted read operation are: read, read_verify and read_loop.

**int read_vme(vme_bus_handle_t handle, vme_component reg)**

The *write_vme()* function writes data stored in the *value_to_write* variable belonging to the *reg* object, to the register pointed to by the *reg* object itself.

**int write_vme(vme_component_reg)**

The *read_dma()* function reads data present in the *reg* DMA register (memory area). The function returns a *vme_component* object. Data will be saved in the *value_read* pointer belonging to the *vme_component* structure.

```
vme_component read_dma(vme_bus_handle_t handle,
                       vme_component reg
```

## 3.4 Functions for executing equipments

An *equipment* is made of *components*. The following functions permit the execution of an *equipment*; this means sequentially performing the operations defined by the *action* field of each *component* belonging to the *equipment* itself.
The *execute_equipment()* function performs all the operations listed in the *eqp equipment* ignoring the *time* variable contained in the *vme_equipment*. Parameters for this function are: the pointer to the bus (*handle*), the *equipment* (*eqp*) and the list of all declared *components* (*component_list*).

```
int execute_equipment(vme_bus_handle_t handle,
vme_component_list component_list, vme_equipment eqp)
```

The *execute_vmelist_equipment()* function executes all *equipments* listed in the *eqp_list* object. Other parameters are: the pointer to the bus (*handle*), the list of all declared *components* (*component_list*), the list of all declared equipments (*eqp_list*), the *index_acquisition* and *time* variables. The *index_acquisition* variable can be 0 or 1. If it is 0 we are in the pre-acquisition phase, otherwise we are in the acquisition phase. The *time* variable, only used during the acquisition phase, represents the time (in seconds) passed since the beginning of data acquisition.

```
int execute_vmelist_equipment(vme_bus_handle_t handle,
vme_component_list component_list, vme_equipment_list eqp_list,
time_t time,int index_acquisition)
```

## 3.5 Functions for executing triggers

A *trigger* is composed by *equipments*. Executing a *trigger* means sequentially performing the operations defined by the *action* field of each *equipment* belonging to the *trigger* itself.
The *execute_trigger()* function performs all the operations defined in a *trigger*. Parameters for this function are: the pointer to the bus (*handle*), the list of all defined *components* (*component_list*) and the *trigger* itself.

**int execute_trigger(vme_bus_handle_t handle, vme_component_list component_list,vme_trigger trig)**

The *execute_vmelist_trigger()* function executes all declared *triggers*. Parameters are: the pointer to the bus (*handle*), the list of all declared *components* (*component_list*), the list of all declared *triggers* (*trig_list*), the *index_acquisition* and *time* variables. The *index_acquisition* variable can be 0 or 1. If it is 0 we are in the pre-acquisition phase, otherwise we are in the acquisition phase. The *time* variable, only used during the acquisition phase, represents the time (in seconds) passed since the beginning of data acquisition.

**int execute_vmelist_trigger(vme_bus_handle_t handle, vme_component_list component_list, vme_trigger_list trig_list, time_t time, int index_acquisition**

## 4 Acquisition run

The DAQ run is managed by the *execute_vmelist_trigger()* function. This function executes all the *triggers* found in the *trig_list* object taking into account the execution period parameter that characterizes each *trigger* in the setup subsection. This execution period parameter is saved in the *time* variable of the *vme_trigger* object. In a typical DAQ system the main program (appendix B) loops on the *execute_vmelist_trigger()* function, updating each time the *time* variable (time passed since the beginning of data acquisition). This value is compared to the execution period parameter of each *trigger* and the decision of executing the *trigger* or not is taken.

## 5 Configuration file interpreter

The whole DAQ system can be controlled by means of a text file (configuration file). In this file all the *components*, *equipments* and *triggers* are defined. The configuration file interpreter reads this file and loads the data structures.

### 5.1 Function for reading the components

The *read_vmelist_components()* function reads the *components* section of the configuration file whose name is read from the *config_file* parameter. The *vme_component_list* structure is loaded.

**vme_component_list ∗read_vmelist_components(char config_file [MAX_LINE_LENGTH])**

### 5.2 Function for reading the equipments

The *load_equipment()* function reads the *equipment* section of the configuration file whose name is read from the *config_file* parameter. The *vme_equipment_list* structure is loaded. Parameters for this function are the list of all declared *components* (*component_list*) and the name of the configuration file (*config_file).*

**vme_equipment_list ∗load_equipment(char config_file[ MAX_LINE_LENGTH],vme_component_list components)**

### 5.3 Function for reading the triggers

The *load_trigger()* function reads the *triggers* section and the setup section of the configuration file whose name is read from the *config_file* parameter. The *vme_trigger_list* structure is loaded. Parameters for this function are the list of all declared *equipments* (*equipments*) and the name of the configuration file (*config_file).*

**vme_trigger_list ∗load_trigger(char config_file[MAX_LINE_LENGTH],vme_equipment_list equipments)**

## References

[1] M. Di Paolo Emilio, S. Stalio *V-DAS (A Versatile Data Acquisition Software): The user interfaces.* LNGS/TC-01/06 - June 2006

# Appendix A: Example of configuration file

```
//example of configuration file
START_COMPONENT_LIST
startcomp ioreg_verify_1
module=V977
register=register_dummy
dw=VME_D16
am=VME_A24UD
base=0xd00000
offset=0x2a
size=1
bus=1
action=write
value=0xdead
expected_value=0
endcomp

startcomp ioreg_verify_2
module=V977
register=register_dummy
dw=VME_D16
am=VME_A24UD
base=0xd00000
offset=0x2a
size=1
bus=1
action=read_verify
value=0
expected_value=0xdead
endcomp

startcomp start_acq
module=V977
register=singlehit
dw=VME_D16
am=VME_A24UD
base=0xd00000
offset=0x16
size=1
bus=1
action=read_loop
```

```
value=
expected_value=0x1
endcomp

END_COMPONENT_LIST

START_EQUIPMENT_LIST

starteqp uno
ioreg_verify_1
ioreg_verify_2
endeqp

starteqp due
start_acq
endeqp

END_EQUIPMENT_LIST

START_TRIGGER_LIST
starttrig tre
uno
due
endtrig
END_TRIGGER_LIST
START_ACQ_SETUP_TRIG
tre -1
END_ACQ_SETUP_TRIG
```

## Appendix B: Example of main program

```
int main()
{
vme_component_list a;
vme_equipment_list b;
vme_trigger_list c;
char file[100]="prova.dat";
int i;
vme_bus_handle_t handle;
time_t t1=0;

a=*read_vmelist_component(file);
b=*load_equipment(file,a);
c=*load_trigger(file,b);
handle=init_bus_vme(handle);

for (i=0;i<a.num_elements;i++)
{
a.component[i]=create_vme_window(handle,a.component[i]);
a.component[i]=map_vme_window(handle,a.component[i]);
}

execute_vmelist_trigger(handle,a,c,t1,0);

while(1)
{
t1=time(0)-t1;
execute_vmelist_trigger(handle,a,c,t1,1);
}
}
```