EDOC024

EUROGAM PROJECT

NSF DATA ACQUISITION SYSTEM

Eurogam Frontend Crate Server

Edition 1.0

September 1990

NSF Software Systems Group

UK Science and Engineering Research Council

Daresbury Laboratory

Eurogam Frontend Crate Server

David Brightly

Edition 1.0
September 1990

1 Introduction

This document elaborates on the description of a "Frontend Crate Server" outlined in [**?**, section 3]. It defines a program model suitable for use in all EuroGam frontend crates supporting Ge cards, BGO cards, histogrammer unit, trigger unit, and apparatus control and environment monitoring interfaces. The program models the electronics in a crate as a set of abstract "registers" and provides a remote procedure call interface to client programs running elsewhere whereby programmable registers can be inspected and updated.

Aspects of the program are described in the Z specification language [**?**]. The specification covers the remote procedure call interface and details of the internal data structures needed to support it. The construction of these data structures by compiling a textual description of a EuroGam hardware configuration is also outlined, together with XDR bindings for the remotely callable procedures. The document concludes with some implementation recommendations and general remarks on the software environment in which the program is expected to run.

2 Crate Management Functions

This section will describe how crates are treated as allocatable resources. We will need to talk about the crates themselves and also about *capabilities*. These are tokens drawn from a large name space, possession of which by a client program confers access rights to a crate. For background to the capability concept see any text on distributed systems, such as [**?**].

$$[CRATE, CAPABILITY]$$

We will consider an environment in which possibly more than one EuroGam system is available, so we will need a set of crates from which a EuroGam can be built:

$$\mid \; crates : \mathbb{P}\, CRATE$$

All client-callable server procedures will return a success or failure report:

$$REPORT ::= ok \mid crate\_not\_known \mid crate\_already\_in\_use \mid capability\_invalid$$

Rather than treat crates individually we will adopt for the time being a God-like viewpoint which observes the state of all the crates. Later on we will add further constraints to these schemas so that they describe single crates. We have a function which maps allocated crates to the capabilities that have been issued to confer access to the crates:

$$
\begin{array}{|l}
\hline
\textit{CrateManager} \\
\hline
\mathit{capabilities} : \mathit{CRATE} \rightarrowtail\!\!\!\!\!\!+\;\; \mathit{CAPABILITY} \\
\hline
\end{array}
$$

Initially, no crates are allocated—this would be the state following a power cut, say.

$$
\begin{array}{|l}
\hline
\textit{InitCrateManager} \\
\hline
\Delta \mathit{CrateManager} \\
\hline
\mathit{capabilities}' = \varnothing \\
\hline
\end{array}
$$

To obtain exclusive access to a crate a client submits a known but unallocated crate and receives in return a capability for that crate. The only constraint on the capability is that it should not currently be associated with an allocated crate.

$$
\begin{array}{|l}
\hline
\textit{ClaimCrateOK} \\
\hline
\Delta \mathit{CrateManager} \\
\mathit{crate}? : \mathit{CRATE} \\
\mathit{capability}! : \mathit{CAPABILITY} \\
\hline
\mathit{crate}? \in \mathit{crates} \\
\mathit{crate}? \notin \operatorname{dom} \mathit{capabilities} \\
\mathit{capability}! \notin \operatorname{ran} \mathit{capabilities} \\
\mathit{capabilities}' = \mathit{capabilities} \oplus \{\mathit{crate}? \mapsto \mathit{capability}!\} \\
\hline
\end{array}
$$

A successful operation returns an *ok* status report:

$$
\begin{array}{|l}
\hline
\textit{EGOk} \\
\hline
\mathit{status}! : \mathit{REPORT} \\
\hline
\mathit{status}! = \mathit{ok} \\
\hline
\end{array}
$$

Operations may fail for several reasons—the crate may not be one of the set known to the system:

$$
\begin{array}{|l}
\hline
\textit{CrateNotKnown} \\
\hline
\Xi \mathit{CrateManager} \\
\mathit{crate}? : \mathit{CRATE} \\
\mathit{status}! : \mathit{REPORT} \\
\hline
\mathit{crate}? \notin \mathit{crates} \\
\mathit{status}! = \mathit{crate\_not\_known} \\
\hline
\end{array}
$$

—it may already be allocated to another client:

$$
\begin{array}{|l}
\hline
\textit{CrateAlreadyInUse} \\
\hline
\Xi \mathit{CrateManager} \\
\mathit{crate}? : \mathit{CRATE} \\
\mathit{status}! : \mathit{REPORT} \\
\hline
\mathit{crate}? \in \operatorname{dom} \mathit{capabilities} \\
\mathit{status}! = \mathit{crate\_already\_in\_use} \\
\hline
\end{array}
$$

—or a submitted capability may not match that associated with a crate (perhaps by accident or possibly through attempted fraud):

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CapabilityInvalid}} \\
\hline
\Xi \textit{CrateManager} \\
\textit{crate}? : \textit{CRATE} \\
\textit{capability}? : \textit{CAPABILITY} \\
\textit{status}! : \textit{REPORT} \\
\hline
\textit{crate}? \in \textit{crates} \\
\textit{capability}? \neq \textit{capabilities}(\textit{crate}?) \\
\textit{status}! = \textit{capability\_invalid} \\
\hline
\end{array}
$$

In none of these failure cases does the state of the system change. The complete *ClaimCrate* operation can be summarised as

$$\textit{ClaimCrate} \mathrel{\widehat=} (\textit{ClaimCrateOk} \land \textit{EGOk}) \lor \textit{CrateNotKnown} \lor \textit{CrateAlreadyInUse}$$

A client can successfully release a crate providing the crate is currently allocated and the client knows the associated capability:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FreeCrateOk}} \\
\hline
\Delta \textit{CrateManager} \\
\textit{crate}? : \textit{CRATE} \\
\textit{capability}? : \textit{CAPABILITY} \\
\textit{status}! : \textit{REPORT} \\
\hline
\textit{crate}? \in \operatorname{dom} \textit{capabilities} \\
\textit{capability}? = \textit{capabilities}(\textit{crate}?) \\
\textit{capabilities}' = \textit{capabilities} \setminus \{\textit{crate}? \mapsto \textit{capability}!\} \\
\textit{status}! = \textit{ok} \\
\hline
\end{array}
$$

Releasing a crate may fail for the additional reason that it is known about but not in use:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CrateNotInUse}} \\
\hline
\Xi \textit{CrateManager} \\
\textit{crate}? : \textit{CRATE} \\
\textit{status}! : \textit{REPORT} \\
\hline
\textit{crate}? \in \textit{crates} \\
\textit{crate}? \notin \operatorname{dom} \textit{capabilities} \\
\textit{status}! = \textit{crate\_not\_in\_use} \\
\hline
\end{array}
$$

The complete *FreeCrate* operation can be characterised as:

$$\textit{FreeCrate} \mathrel{\widehat=} (\textit{FreeCrateOk} \land \textit{EGOk}) \lor \textit{CrateNotInUse} \lor \textit{CrateNotKnown} \lor \textit{CapabilityInvalid}$$

3 Specifying a EuroGam Configuration

Earlier papers have posited a software interface to the EuroGam frontend crates couched in terms of abstract registers which are mapped onto physical hardware registers in modules. This section will describe the logical "infrastructure" of sets and functions needed to support the mapping. A subsequent section will show how these sets and functions are constructed.

3.1 Given sets

We will need some "given sets" for our specification. These are essentially the "types" of the objects that the specification will talk about.

$$[ARRAYPOSITION]$$

Objects of type *ARRAYPOSITION* will denote the "holes" in the array into which detectors are inserted. They will be names like G23, denoting a Ge detector position, and S23d, denoting position d in the surrounding BGO shield.

$$[DETECTOR]$$

Objects of type *DETECTOR* will denote actual detectors. They will be names like GUOC17 and SUC11.

$$[DETECTORTYPE]$$

Types of detectors, typically either Ge or BGO.

$$[MODULE]$$

Objects of type *MODULE* will denote physical modules such as VXI Ge cards that can occupy slots in crates. Modules will be individually identified by (say) serial numbers such as G#003, denoting the Ge card with serial number 003.

$$MODULETYPE ::= G \mid B \mid T \mid H \mid RC \mid RM$$

Types of modules used in EuroGam such as Ge cards, BGO cards, Trigger units, Histogrammers, Readout Controllers, and Resource Managers.

$$[SOCKET]$$

Objects of type *SOCKET* will represent physical labelled sockets on the front panels of individual modules into which cables can be inserted. Different modules will have different sockets even if the modules are of the same type. For example, G#003.BNC1 would refer to BNC socket number 1 on Ge module G#003.

$$[SUBCHANNEL]$$

Ge and BGO modules consist of several similar independent subsystems each serving a single detector. A *SUBCHANNEL* will be a small integer denoting one such channel.

$$[CRATE]$$

Objects of type *CRATE* will represent the individual crates comprising a EuroGam system, such as VXI1.

$[NETADDR]$

Objects of type *NETADDR* will denote network addresses for the Resource Manager modules associated with each crate. These addresses may be IP addresses or host names—it doesn't matter for the purpose of the specification.

$[SLOT]$

Objects of type *SLOT* will represent slots in crates for modules. For example, VXI1.N2 will denote slot 2 of crate VXI1. Note that different crates have different slots.

$[REGISTER]$

Objects of type *REGISTER* will denote abstract names for the software addressable registers of modules such as G23.PoleZero, G17.E1, and Trigger.TimingWindow. Different modules will have different registers even if of the same type.

$[PROPERTY]$

Objects of type *PROPERTY* will represent the abstract properties of EuroGam modules, such CFD levels, timing windows, and polezero adjusters with which registers are associated. The set of properties can be completed only when the register sets of each type of module have been finalised.

$[VMEADDRESS]$

Objects of type *VMEADDRESS* will be integers denoting VME addresses.

3.2 Fixed functions

The following functions are independent of any particular EuroGam configuration—they merely extract information implicit in the objects they take as arguments.

$$\begin{array}{|l}
dtype : DETECTOR \rightarrow DETECTORTYPE \\
mtype : MODULE \rightarrow MODULETYPE \\
incrate : SLOT \rightarrow CRATE \\
onmodule : SOCKET \rightarrow MODULE \\
feeds : SOCKET \rightarrow SUBCHANNEL \\
atposition : REGISTER \nrightarrow ARRAYPOSITION \\
ofdetector : REGISTER \nrightarrow DETECTOR \\
property : REGISTER \rightarrow PROPERTY \\
supportedby : PROPERTY \rightarrow MODULETYPE
\end{array}$$

dtype tells what type any detector is. For example,

$$dtype(\text{GUOC17}) = \text{Ge}$$

mytpe tells what type any module is. For example,

$$mtype(\text{G\#003}) = \text{G}$$

incrate tells which crate any slot belongs to. For example,

$$incrate(\text{VXI1.N3}) = \text{VXI1}$$

onmodule tells which module's front panel a socket is on. For example,

$$onmodule(\text{G\#003.BNC1}) = \text{G\#003}$$

feeds tells which subchannel of a module a given socket feeds into. For example, perhaps

$$feeds(\text{G\#003.BNC1}) = 1$$

atposition tells which detector position a register refers to. For example,

$$atposition(\text{G23.Bias}) = \text{G23}$$

ofdetector tells which detector a register refers to. For example,

$$ofdetector(\text{GUOC17.Bias}) = \text{GUOC17}$$

property tells what "type" each abstract register is. For example,

$$property(\text{G23.Bias}) = \text{Bias}$$

supportedby relates properties and module types. For example,

$$supportedby(\text{PoleZero}) = \text{G}$$

We have *atposition* because we want to name the registers after the data acquisition channels they control rather than use labels based on VXI crate and slot positions which won't mean much to users. However, not all registers will refer to array positions. Some will refer to aspects of Trigger, Histogrammer, or other modules. We also want registers that refer to detectors so that, for example, detector biases can be set in a natural way. Hence the *ofdetector* function.

3.3 A EuroGam configuration

We will represent any particular EuroGam configuration with a collection of sets of objects drawn from the above types and some functions between them:

EuroGam

$$
\begin{array}{l}
holes : \mathbb{P}\, ARRAYPOSITION \\
detectors : \mathbb{P}\, DETECTOR \\
crates : \mathbb{P}\, CRATE \\
modules : \mathbb{P}\, MODULE \\
slots : \mathbb{P}\, SLOT \\
sockets : \mathbb{P}\, SOCKET \\
registers : \mathbb{P}\, REGISTER \\
triggers : \mathbb{P}\, MODULE \\
histogrammers : \mathbb{P}\, MODULE
\end{array}
$$

$$
\begin{array}{|l}
sitsin : DETECTOR \pinj ARRAYPOSITION \\
occupies : MODULE \pinj SLOT \\
basedat : SLOT \pfun VMEADDRESS \\
hostedby : CRATE \inj NETADDRESS \\
wiredto : ARRAYPOSITION \pinj SOCKET \\
biasedby : ARRAYPOSITION \pinj SOCKET \\
\hline
detectors = \operatorname{dom} sitsin \\
holes = \operatorname{ran} sitsin \\
modules = \operatorname{dom} occupies \\
slots = \operatorname{ran} occupies \subseteq \operatorname{dom} basedat \\
incrate \llparenthesis slots \rrparenthesis \subseteq crates \\
slots \subseteq \operatorname{dom} basedat \\
onmodule \llparenthesis sockets \rrparenthesis \subseteq modules \\
wiredto \llparenthesis holes \rrparenthesis \subseteq sockets \\
atposition \llparenthesis registers \rrparenthesis = \operatorname{dom} wiredto \cup \operatorname{dom} biasedby \\
\hline
\end{array}
$$

holes is the set of occupied detector positions in this particular EuroGam; this will depend on the mechanical configuration of the array and detectors.

detectors is the set of detectors actually mounted in the array.

sitsin is a function which tells which position each and every detector occupies.

crates is the set of electronics crates associated with the array.

modules is the set of modules in the crates.

occupies is a function telling which slot is taken by each module.

slots is just the set of the occupied slots in crates.

basedat is a function returning the VME base address of each occupied slot.

sockets is the set of front panel sockets on all the modules in the crates.

wiredto is a function playing a central role in the configuration, telling which VXI module socket every detector position is cabled to.

biasedby is a similar function to *wiredto*, telling which high voltage bias socket each detector position is connected to.

registers is the set of (names of) software controllable registers in the modules.

Note that *wiredto* and *biasedby* map array positions rather than actual detectors to module sockets. This is because the array cabling will be relatively static—it's the detectors that will move in and out of the array—and *wiredto* and *biasedby* capture this relationship.

3.4 Implementing register access

This section will show how abstract registers are mapped onto physical VME addresses by in-crate software. Essentially, there are three classes of register. Firstly we have registers referring to specific array positions:

$$
\begin{array}{|l}
\hline
\textit{LocateArrayRegister} \\
\Xi \textit{EuroGam} \\
r? : \textit{REGISTER} \\
\textit{modtype}! : \textit{MODULETYPE} \\
\textit{crate}! : \textit{CRATE} \\
\textit{addr}! : \textit{VMEADDRESS} \\
\textit{prop}! : \textit{PROPERTY} \\
\textit{subchan}! : \textit{SUBCHANNEL} \\
\hline
r? \in \operatorname{dom} \textit{atposition} \\
\textit{prop}! = \textit{property}(r?) \\
\textit{modtype}! = \textit{mtype} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{atposition}(r?) \\
\textit{crate}! = \textit{incrate} \circ \textit{occupies} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{atposition}(r?) \\
\textit{addr}! = \textit{basedat} \circ \textit{occupies} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{atposition}(r?) \\
\textit{subchan}! = \textit{feeds} \circ \textit{wiredto} \circ \textit{atposition}(r?) \\
\hline
\end{array}
$$

Secondly, we have registers referring to the trigger unit, histogrammer unit, and possibly others:

$$
\begin{array}{|l}
\hline
\textit{LocateGeneralRegister}[X] \\
\Xi \textit{EuroGam} \\
r? : \textit{REGISTER} \\
\textit{modtype}! : \textit{MODULETYPE} \\
\textit{crate}! : \textit{CRATE} \\
\textit{addr}! : \textit{VMEADDRESS} \\
\textit{prop}! : \textit{PROPERTY} \\
\textit{subchan}! : \textit{SUBCHANNEL} \\
\hline
\exists\, m : X \bullet \\
\quad \textit{supportedby} \circ \textit{property}(r?) = \textit{mtype}(m) \\
\quad \textit{prop}! = \textit{property}(r?) \\
\quad \textit{modtype}! = \textit{mtype}(m) \\
\quad \textit{crate}! = \textit{incrate} \circ \textit{occupies}(m) \\
\quad \textit{addr}! = \textit{basedat} \circ \textit{occupies}(m) \\
\quad \textit{subchan}! = 0 \\
\hline
\end{array}
$$

Finally, we have registers that refer to particular detectors:

$$
\begin{array}{|l}
\hline
\textit{LocateDetectorRegister} \\
\Xi \textit{EuroGam} \\
r? : \textit{REGISTER} \\
\textit{modtype}! : \textit{MODULETYPE} \\
\textit{crate}! : \textit{CRATE} \\
\textit{addr}! : \textit{VMEADDRESS} \\
\textit{prop}! : \textit{PROPERTY} \\
\textit{subchan}! : \textit{SUBCHANNEL} \\
\hline
r? \in \operatorname{dom} \textit{ofdetector} \\
\textit{ofdetector}(r?) \in \operatorname{dom} \textit{sitsin} \\
\textit{prop}! = \textit{property}(r?) \\
\textit{modtype}! = \textit{mtype} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{sitsin} \circ \textit{ofdetector}(r?) \\
\textit{crate}! = \textit{incrate} \circ \textit{occupies} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{sitsin} \circ \textit{ofdetector}(r?) \\
\textit{addr}! = \textit{basedat} \circ \textit{occupies} \circ \textit{onmodule} \circ \textit{wiredto} \circ \textit{sitsin} \circ \textit{ofdetector}(r?) \\
\textit{subchan}! = \textit{feeds} \circ \textit{wiredto} \circ \textit{sitsin} \circ \textit{ofdetector}(r?) \\
\hline
\end{array}
$$

Combining these, we can locate an arbitrary register:

$$
\begin{array}{l}
\textit{LocateRegister} \mathrel{\widehat=} \textit{LocateArrayRegister} \lor \textit{LocateGeneralRegister}[\textit{triggers}] \\
\quad \lor \textit{LocateGeneralRegister}[\textit{histogrammers}] \lor \textit{LocateDetectorRegister}
\end{array}
$$

These schemas assume that to access the hardware we need to know five things:

- the "type" of the register;
- the type of the module in which the register resides;
- the crate housing this module;
- the module's VME base address;
- which subchannel of the module the register applies to.

Note that the constraints on the sets and functions implied by *EuroGam* guarantee that these outputs are defined.

We will assume that registers hold typed pieces of data drawn from the set

$$[\textit{VALUE}]$$

and that we have a given relation on values and register properties that tells us when they are type-comaptible:

$$
\begin{array}{|l}
\textit{compatible} : \textit{VALUE} \leftrightarrow \textit{PROPERTY}
\end{array}
$$

A *Read* operation returns such a value from the hardware:

$$
\begin{array}{|l}
\hline
\textit{Read} \\
\Xi \textit{EuroGam} \\
\textit{modtype}? : \textit{MODULETYPE} \\
\textit{crate}? : \textit{CRATE} \\
\textit{addr}? : \textit{VMEADDRESS} \\
\textit{prop}? : \textit{PROPERTY} \\
\textit{subchan}? : \textit{SUBCHANNEL} \\
\textit{value}! : \textit{VALUE} \\
\hline
\end{array}
$$

and a *Write* operation takes a value and applies it to the hardware:

$$
\begin{array}{|l}
\hline
\textit{Write} \\
\Xi \textit{EuroGam} \\
\textit{modtype}? : \textit{MODULETYPE} \\
\textit{crate}? : \textit{CRATE} \\
\textit{addr}? : \textit{VMEADDRESS} \\
\textit{prop}? : \textit{PROPERTY} \\
\textit{subchan}? : \textit{SUBCHANNEL} \\
\textit{value}? : \textit{VALUE} \\
\hline
\end{array}
$$

We will refrain from making any statements about what the hardware system does with the data. For example, it need not always be the case that

$$\textit{Write} \mathbin{⨾} \textit{Read} \Rightarrow \textit{value}! = \textit{value}?$$

since we are warned not to expect all hardware registers to be readable.

In practice we would associate a procedure or package of procedures with module type *modtype*! which could determine a register's VME address given the *addr*!, *prop*!, and *subchan*! arguments. Such a procedure package would have to "know" all about the internal register structure of modules of this type.

4 Register Access Functions

This section will describe operations based on the abstract registers introduced previously. First, a schema encapsulating the crate access protection mechanism which we will use to abbreviate the descriptions:

$$
\begin{array}{|l}
\hline
\textit{CrateAccess} \\
\Xi \textit{CrateManager} \\
\textit{crate}? : \textit{CRATE} \\
\textit{capability}? : \textit{CAPABILITY} \\
\hline
\textit{capability}? = \textit{capabilities}(\textit{crate}?) \\
\hline
\end{array}
$$

To read a register successfully a client must submit a valid crate and corresponding capability, and a register which resides in that crate. The value currently held in the register is returned.

$$
\begin{array}{|l}
\hline
\textit{ReadRegisterOk} \\
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{register}? : \textit{REGISTER} \\
\textit{value}! : \textit{VALUE} \\
\hline
\textit{register}? \in \textit{registers} \\
\textit{LocateRegister} \mathbin{⨾} \textit{Read} \\
\hline
\end{array}
$$

Writing a register successfully requires the client to submit in addition a value which is type-compatible with the register. The register is updated with the value submitted.

$$
\begin{array}{|l}
\hline
\textit{WriteRegisterOk} \\
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{register}? : \textit{REGISTER} \\
\textit{value}? : \textit{VALUE} \\
\hline
\textit{register}? \in \textit{registers} \\
(\textit{value}?, \textit{property}(\textit{register}?)) \in \textit{compatible} \\
\textit{LocateRegister} \mathbin{;} \textit{Write} \\
\hline
\end{array}
$$

Initialising a register is like writing it except that the value comes from the server not the client. We will assume that for each register the server knows a safe value. In practice, it is likely that registers sharing the same property will also share the same safe value.

$$
\begin{array}{|l}
\textit{safe} : \textit{REGISTER} \rightarrow \textit{VALUE}
\end{array}
$$

$$
\textit{InitialiseRegisterOk} \mathrel{\widehat=} \textit{WriteRegisterOk}[\textit{safe}(\textit{register}?)/\textit{value}?]
$$

The next operations use a pattern argument to select registers. For the time being we will regard the pattern matching as a given relation between registers and patterns:

$$
[\textit{PATTERN}]
$$

$$
\begin{array}{|l}
\textit{matches} : \textit{REGISTER} \leftrightarrow \textit{PATTERN}
\end{array}
$$

The simplest pattern matching operation returns the set of registers that match a submitted pattern:

$$
\begin{array}{|l}
\hline
\textit{InquireRegistersOk} \\
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{pattern}? : \textit{PATTERN} \\
\textit{rlist}! : \mathbb{P}\, \textit{REGISTER} \\
\hline
\forall r : \textit{REGISTER} \mid r \in \textit{registers} \bullet \\
\quad r \in \textit{rlist}! \Leftrightarrow (r, \textit{pattern}?) \in \textit{matches} \\
\hline
\end{array}
$$

Next, we submit a pattern and are returned a set of pairs of registers and their corresponding values:

$$
\begin{array}{|l}
\hline
\textit{ReadRegistersOk} \\
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{pattern}? : \textit{PATTERN} \\
\textit{rvlist}! : \mathbb{P}\, \textit{REGISTER} \times \textit{VALUE} \\
\hline
\forall r : \textit{REGISTER};\ v : \textit{VALUE} \mid r \in \textit{registers} \bullet \\
\quad (r, v) \in \textit{rvlist}! \Leftrightarrow \\
\qquad (r, \textit{pattern}?) \in \textit{matches} \land \\
\qquad\quad (\textit{LocateRegister} \mathbin{;} \textit{Read})[v/\textit{value}!, r/\textit{register}?] \\
\hline
\end{array}
$$

We can write a single value to a set of registers that match a pattern and are type-compatible with the submitted value:

$$
\begin{array}{|l}
\hline
\textit{WriteRegistersOk} \\
\hline
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{pattern}? : \textit{PATTERN} \\
\textit{value}? : \textit{VALUE} \\
\hline
\forall r : \textit{REGISTER} \mid r \in \textit{registers} \bullet \\
\quad (r, \textit{pattern}?) \in \textit{matches} \land (\textit{value}?, \textit{property}(r)) \in \textit{compatible} \Rightarrow \\
\qquad (\textit{LocateRegister} \fatsemi \textit{Write})[r / \textit{register}?] \\
\hline
\end{array}
$$

And we can load safe values into all registers matching a pattern:

$$
\begin{array}{|l}
\hline
\textit{InitialiseRegistersOk} \\
\hline
\Xi \textit{EuroGam} \\
\textit{CrateAccess} \\
\textit{pattern}? : \textit{PATTERN} \\
\textit{value}? : \textit{VALUE} \\
\hline
\forall r : \textit{REGISTER} \mid r \in \textit{registers} \bullet \\
\quad (r, \textit{pattern}?) \in \textit{matches} \Rightarrow \\
\qquad (\textit{LocateRegister} \fatsemi \textit{Write})[\textit{safe}(r) / \textit{value}?, r / \textit{register}?] \\
\hline
\end{array}
$$

Operations specifying registers by name rather than by a pattern can fail because the register submitted is unknown:

$$
\begin{array}{|l}
\hline
\textit{RegisterNotKnown} \\
\hline
\Xi \textit{EuroGam} \\
\textit{register}? : \textit{REGISTER} \\
\textit{status}! : \textit{REPORT} \\
\hline
\textit{register}? \notin \textit{registers} \\
\textit{status}! = \textit{register\_not\_known} \\
\hline
\end{array}
$$

A specific write operation may fail because of a type incompatibility:

$$
\begin{array}{|l}
\hline
\textit{TypesIncompatible} \\
\hline
\Xi \textit{EuroGam} \\
\textit{register}? : \textit{REGISTER} \\
\textit{value}? : \textit{VALUE} \\
\textit{status}! : \textit{REPORT} \\
\hline
\textit{register}? \in \textit{registers} \\
(\textit{value}?, \textit{property}(\textit{register}?)) \notin \textit{compatible} \\
\textit{status}! = \textit{types\_incompatible} \\
\hline
\end{array}
$$

Finally, we can describe robust register access functions:

$$
\begin{aligned}
&\mathit{ReadRegister} \mathrel{\widehat=} (\mathit{ReadRegisterOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}\\
&\quad \lor \mathit{RegisterNotKnown}\\
&\mathit{WriteRegister} \mathrel{\widehat=} (\mathit{WriteRegisterOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}\\
&\quad \lor \mathit{RegisterNotKnown} \lor \mathit{TypesIncompatible}\\
&\mathit{InitialiseRegister} \mathrel{\widehat=} (\mathit{InitialiseRegisterOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}\\
&\quad \lor \mathit{RegisterNotKnown}\\
&\mathit{ReadRegisters} \mathrel{\widehat=} (\mathit{ReadRegistersOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}\\
&\mathit{WriteRegisters} \mathrel{\widehat=} (\mathit{WriteRegistersOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}\\
&\mathit{InitialiseRegisters} \mathrel{\widehat=} (\mathit{InitialiseRegistersOk} \land \mathit{EGOk}) \lor \mathit{CrateNotInUse} \lor \mathit{CapabilityInvalid}
\end{aligned}
$$

5 Establishing a Configuration

Each crate has to contain the data structures implied by the *EuroGam* schema in order to support register access as outlined in the previous section. A simple way of achieving this is to represent the sets and functions in some sort of configuration description file. The frontend software is required to compile the description into suitable data structures for implementing the *LocateRegister* schema. We would expect a EuroGam configuration to change infrequently—perhaps at the start of an experiment or following replacement of a faulty module. In these circumstances it is difficult to see how a partly established set up of register values can be preserved, since crates are likely to have been power-cycled and module and wiring configuration altered. Hence we must accept that compiling a configuration from a description file will destroy all previously established information.

We will need a file in which to record diagnostics (such as syntax errors) while compiling the description:

$$
\begin{array}{|l}
\hline
\mathit{EuroGamLog} \\
\mathit{log} : \mathit{FILE} \\
\hline
\end{array}
$$

We need not be very specific about the log file contents as long as it consists of readable text that helps pinpoint problems with the configuration. We would like to be able to read it back at any time though:

$$
\begin{array}{|l}
\hline
\mathit{ReadEuroGamLog} \\
\Xi \mathit{EuroGamLog} \\
\mathit{file}! : \mathit{FILE} \\
\hline
\mathit{file}! = \mathit{log} \\
\hline
\end{array}
$$

The *ConfigureEuroGam* operation takes a description file, compiles a new configuration, and updates the log:

$$
\begin{array}{|l}
\hline
\mathit{ConfigureEuroGam} \\
\Delta \mathit{EuroGam} \\
\Delta \mathit{EuroGamLog} \\
\mathit{description}? : \mathit{FILE} \\
\hline
\end{array}
$$

A suitable form for a description file follows quite naturally from the signature of the *EuroGam* schema.

EGspec	::=	cratespec ... cratespec detectorspec ... detectorspec	
cratespec	::=	`crate` *crate* `host` *host* [modulespec ... modulespec]	
modulespec	::=	`module` *module* `slot` *slot* [socketspec ... socketspec]	
socketspec	::=	`socket` *socket* `wiredfrom` *position*	– data connection
	|	`socket` *socket* `biases` *position*	– HV connection
detectorspec	::=	`detector` *detector* `position` *position*	

For example:

```
crate VXI1 host NNVA
module G#003 slot 4
socket BNC1 wiredfrom G23
socket BNC2 wiredfrom G24
socket BNC3 wiredfrom G25
module B#007 slot 5
socket IDC1.1 wiredfrom S23a
socket IDC1.2 wiredfrom S23b
module T#002 slot 3
detector GUOC17 position G23
```

This describes a crate called VXI1 with network address NNVA. It has a Ge module, serial #003, in slot 4. BNC socket 1 of this module connects to array position G23, socket 2 to position G24, and socket 3 to position G25. In slot 5 of the crate is a BGO module, serial #007. It has an IDC connector IDC1 with pin 1 connected to shield detector position S23a and pin 2 to S23b. Trigger unit serial #002 occupies slot 3. Germanium detector GUOC17 sits in array position G23. We can exploit the expected regularity of the structure and cabling with special pattern-matching syntax, using, say

```
socket BNC[1-3] wiredfrom G[23-25]
```

to abbreviate the first three `socket` lines.

Note that we are using the natural structure of "sockets are on modules which occupy slots in crates" to save duplicating information. The "proper name" of the slot occupied by module G#003 is VXI1.N4 and that of its first socket is G#003.BNC1. The syntax of the description file guarantees that the proper names for these objects can always be recovered.

Compiling a description is straightforward. We start in an initial state in which we know nothing at all:

InitEuroGam

$\Delta EuroGam$

$$holes' = \varnothing \wedge detectors' = \varnothing \wedge crates' = \varnothing$$
$$modules' = \varnothing \wedge slots' = \varnothing \wedge sockets' = \varnothing$$
$$registers' = \varnothing \wedge sitsin' = \varnothing \wedge occupies' = \varnothing$$
$$hostedby' = \varnothing \wedge wiredto' = \varnothing \wedge biasedby' = \varnothing$$

The description file is trivially parsed and the full configuration is built up piece by piece. The following schema applies as soon as a *host* has been read while parsing a cratespec:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InstallCrate}} \\
\Delta \mathit{EuroGam} \\
\mathit{crate?} : \mathit{CRATE} \\
\mathit{host?} : \mathit{NETADDRESS} \\
\hline
\mathit{crates}' = \mathit{crates} \cup \{\mathit{crate?}\} \\
\mathit{hostedby}' = \mathit{hostedby} \oplus \{\mathit{crate?} \mapsto \mathit{host?}\}
\end{array}
$$

Similarly, *InstallModule* applies once a *slot* has been found when parsing a modulespec. The *slot*? input is constructed from *slot* and its immediately preceeding *crate*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InstallModule}} \\
\Delta \mathit{EuroGam} \\
\mathit{crate?} : \mathit{CRATE} \\
\mathit{module?} : \mathit{MODULE} \\
\mathit{slot?} : \mathit{SLOT} \\
\hline
\mathit{incrate}(\mathit{slot?}) = \mathit{crate?} \\
\mathit{occupies}' = \mathit{occupies} \oplus \{\mathit{module?} \mapsto \mathit{slot?}\}
\end{array}
$$

General modules (i.e. other than Ge and BGO types) introduce a set of registers when they are installed, and they should be installed only once! For example:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InstallTriggerModule}} \\
\Delta \mathit{EuroGam} \\
\mathit{module?} : \mathit{MODULE} \\
\hline
\mathit{mtype}(\mathit{module?}) = \mathrm{T} \\
\mathit{triggers} = \varnothing \\
\mathit{triggers}' = \{\mathit{module?}\} \\
\mathit{registers}' = \mathit{registers} \cup \\
\qquad \{r : \mathit{REGISTER} \mid \mathit{supportedby} \circ \mathit{property}(r) = \mathit{mtype}(\mathit{module?})\}
\end{array}
$$

Detectors can be installed when a detectorspec has been parsed:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InstallDetector}} \\
\Delta \mathit{EuroGam} \\
\mathit{detector?} : \mathit{DETECTOR} \\
\mathit{position?} : \mathit{ARRAYPOSITION} \\
\hline
\mathit{detector?} \notin \operatorname{dom} \mathit{sitsin} \\
\mathit{position?} \notin \operatorname{ran} \mathit{sitsin} \\
\mathit{sitsin}' = \mathit{sitsin} \oplus \{\mathit{detector?} \mapsto \mathit{position?}\} \\
\mathit{detectors}' = \mathit{detectors} \cup \{\mathit{detector?}\} \\
\mathit{holes}' = \mathit{holes} \cup \{\mathit{position?}\}
\end{array}
$$

A socket can be installed once a complete socketspec has been parsed. Sockets can terminate data connections and bias supply connections:

$$
\begin{array}{l}
\textbf{InstallDataSocket} \\
\hline
InstallSocket \\
\hline
wiredto' = wiredto \oplus \{position? \mapsto socket?\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{InstallHVSocket} \\
\hline
InstallSocket \\
\hline
biasedby' = biasedby \oplus \{position? \mapsto socket?\}
\end{array}
$$

where we have defined

$$
\begin{array}{l}
\textbf{InstallSocket} \\
\hline
\Delta EuroGam \\
crate? : CRATE \\
module? : MODULE \\
socket? : SOCKET \\
position? : ARRAYPOSITION \\
\hline
incrate \circ occupies(module?) = crate? \\
onmodule(socket?) = module? \\
sockets' = sockets \cup \{socket?\} \\
holes' = holes \cup \{position?\} \\
registers' = registers \cup \\
\quad \{r : REGISTER \mid atposition(r) = position? \land \\
\quad\quad supportedby \circ property(r) = mtype(module?)\}
\end{array}
$$

Again, *socket?* is constructed from *socket* and its immediately preceeding *module*. Note how installing a socket extends the set of registers we know about by just those registers associated with the array position connected up to the socket.

These operations all maintain the predicate of the *EuroGam* schema.

6 XDR Bindings

This section details XDR type definitions and procedure bindings for the operations specified in the previous section. For background information on XDR see [**?**].

First, a set of `typedef`s for objects drawn from our given sets:

```
typedef string crate<255>;
typedef string register<255>;
typedef string pattern<255>;
typedef string filename<255>;
typedef string bytestring<255>;
typedef opaque capability[4];
typedef opaque cookie[4];

enum basictype {
   RV_INT = 1;
```

```
      RV_FLOAT = 2;
      RV_BYTESTRING = 3;
      RV_STRING = 4;
      RV_BOOL = 5;
      };

   union registervalue switch (basictype rvtype) {
      case RV_INT:        int i;
      case RV_FLOAT:      float f;
      case RV_BYTESTRING: bytestring bs;
      case RV_STRING:     string s;
      case RV_BOOL:       bool b;
      };

   enum report {
      EG_OK = 0;
      EG_CRATE_NOT_KNOWN = 1;
      EG_CRATE_ALREADY_IN_USE = 2;
      EG_CAPABILITY_INVALID = 3;
      EG_REGISTER_NOT_KNOWN = 4;
      EG_TYPES_INCOMPATIBLE = 5;
      }
```

All the operations we need return a report parameter indicating the success of failure of the operation. Only if the operation succeeds will further parameters be returned. Hence all operation outputs are coded in the form:

```
   union reply switch (report r) {
      case EG_OK:   struct success;
      default: void;
      };
```

where the `success` structure depends on the particular procedure invoked. To save space, in the XDR bindings that follow we will give just the `success` structures. The syntax follows that of an earlier version of XDR which is a little more compact and looks more like function declarations in the C language.

```
   procedure ClaimCrate (c) returns (reply)
      crate c;
      struct success {capability cap;};

   procedure FreeCrate (cap, c) returns (reply)
      capability cap;
      crate c;
      void success;

   procedure ConfigureEuroGam (cap, c, f) returns (reply)
      capability cap;
      crate c;
```

```
   filename f;
   void success;

procedure ReadEuroGamLog (cap, c) returns (reply)
   capability cap;
   crate c;
   struct success {string file;};

procedure ReadRegister (cap, c, r) returns (reply)
   capability cap;
   crate c;
   register r;
   struct success {registervalue v;};

procedure WriteRegister (cap, c, r, v) returns (reply)
   capability cap;
   crate c;
   register r;
   registervalue v;
   void success;

procedure InitialiseRegister (cap, c, r) returns (reply)
   capability cap;
   crate c;
   register r;
   void success;

typedef struct {
   register r;
   cookie kk;
   reglist *rr;
   } reglist;

typedef struct {
   register r;
   registervalue v;
   cookie kk;
   regvallist *rr;
   } regvallist;

procedure InquireRegisters (cap, c, p, k, n) returns (reply)
   capability cap;
   crate c;
   pattern p;
   cookie k;
   int n;
   struct success {
```

```
        reglist *regs;
        boolean eol;
        };

procedure ReadRegisters (cap, c, p, k, n) returns (reply)
    capability cap;
    crate c;
    pattern p;
    cookie k;
    int n;
    struct success {
        regvallist *regvals;
        boolean eol;
        };

procedure WriteRegisters (cap, c, p, v) returns (reply)
    capability cap;
    crate c;
    pattern p;
    registervalue v;
    void success;

procedure InitialiseRegisters (cap, c, p) returns (reply)
    capability cap;
    crate c;
    pattern p;
    void success;
```

These are all straightforward XDR definitions apart from `InquireRegisters` and `ReadRegisters` which are potentially capable of returning unbounded sets of parameters. We use the technique outlined in [?, page 179] to obtain the full set of parameters in one or more chunks. `InquireRegisters` returns a variable length list of register names, upto a maximum of `n` per call, each matching the pattern `p`. Each register is accompanied by a "cookie", `kk`, which is an opaque pointer to the next register in the list. Any cookie can be used in a subsequent `InquireRegisters` call to obtain further register names starting with the one identified by the cookie. To get the list started a special cookie value with all bits zero identifies the first entry in the list of matches. Finally, the boolean `eol` takes the value `TRUE` when there are no more matching registers in the list. `ReadRegisters` is very similar, except that each matching register name is also accompanied by a value. The encodings for the procedures themselves are:

```
program FrontendCrateServer {
    version FCSversion {
        procedure ClaimCrate = 1;
        procedure FreeCrate = 2;
        procedure ConfigureEuroGam = 3;
        procedure ReadEuroGamLog = 4;
        procedure ReadRegister = 5;
        procedure WriteRegister = 6;
```

```
    procedure InitialiseRegister = 7;
    procedure InquireRegisters = 8;
    procedure ReadRegisters = 9;
    procedure WriteRegisters = 10;
    procedure InitialiseRegisters = 11;
    } = 1;
} = 0x2000001;
```

7 Implementation Recommendations

7.1 Crate data structures

The subset of a EuroGam configuration corresponding to a single crate can be represented as a further constrained EuroGam configuration. For example:

$$\begin{array}{|l}
\textit{EuroGamCrate} \\
\textit{EuroGam} \\
\hline
\textit{crates} = \{\text{VXI1}\} \\
\hline
\end{array}$$

We can arrive at a model for the data structures needed in a particular EuroGam crate by restricting the *EuroGam* sets and functions to the given crate. When compiling a description file, for example, all cratespecs not referring to the given crate are simply ignored. The set *registers* ends up containing just those registers which "live" in the given crate.

7.2 Representing sets and functions

All the basic objects we need are essentially names or 'symbols'. This suggests that an implementation should build a symbol table and use fixed length (e.g. 32 bit) symbol table indexes or pointers to represent basic objects. A set of objects can then be represented by a list of such pointers, and a function f can be represented as a list of pairs of pointers (x, y) such that $f(x) = y$. Evaluating a function involves first finding the symbol table entry, x, for the argument symbol, scanning the list of pairs representing the function for a pair whose first element equals x, and then returning the corresponding y. Finally the pointer y can be used to probe the symbol table for the result symbol. Pattern matching a name against some set involves working down the list of pointers representing the set, probing the symbol table for the corresponding name and applying the pattern matching procedure. Since we can expect to be dealing with some tens of thousands of basic objects the symbol table should be organised as a balanced binary tree ordered lexicographically on the symbols. Functions and sets can also be represented by balanced binary trees, this time ordered by the pointer values themselves (the x pointers in the case of functions). This organisation is preferable to fixed sized arrays and tables, being much more flexible and adaptable.

7.3 Generating capabilities

For this application we need to ensure against accidental rather than malicious interference between distinct sets of clients. Expensively encrypted capabilities are therefore waived in favour of a cheap solution that guards against a client accidentally hitting upon a valid capability value. It is important that capability values recycle slowly (periods of several days are desirable) and do not recycle as a result of re-booting crates. A scheme based on a crate processor's real-time clock is recommended.

7.4 Pattern matching on registers

We will assume that register names are made from just the upper and lower case letters, the digits, and the period. The most useful patterns for this application are a wildcard pattern which matches zero or more characters and numeric and character range patterns that match any integer or character in a sequence of consecutive integers or characters.

```
pattern ::= pat ... pat
pat     ::= letter
          | .
          | digit
          | *
          | [range, ... ,range]
range   ::= integer
          | integer-integer
          | letter
          | letter-letter
integer ::= digit ... digit
```

where letter denotes any of the upper and lower case letters and digit any of the digits 0 to 9. For example, the pattern `[5,10-12,a-d,19]` matches any of the set $\{5, 10, 11, 12, a, b, c, d, 19\}$, and the pattern `G[1-2].PoleZero` matches G1.PoleZero and G2.PoleZero. The wildcard pattern `G23.*` matches all names beginning G23. and followed by zero or more characters, i.e., all registers related to array position G23.

7.5 Calibration functions

Many of the analogue properties of, for example, Ge modules are more conveniently expressed in "natural" units such as microseconds and millivolts, rather than the integer representations (0 to 255, say) necessarily required by the hardware in most cases. This requires two functions:

$$
\begin{array}{|l}
\mathit{writecalibration} : \mathit{REGISTER} \nrightarrow \mathit{VALUE} \nrightarrow \mathbb{Z} \\
\mathit{readcalibration} : \mathit{REGISTER} \nrightarrow \mathbb{Z} \nrightarrow \mathit{VALUE}
\end{array}
$$

which, given a register, return an output and an input calibration function for that register, respectively. At this stage of the hardware design it is not yet clear what level of support for calibrations is required:

- do we need, for each analogue property, distinct calibrations for each individual register, or for each module, or will a single fixed function for all modules suffice?
- to what extent should it be possible to alter these functions dynamically?

If a single, immutable function (or pair of functions) for each property is adequate then these can be built into the crate server program as "given" functions. This is the simplest solution, and devoutly to be desired. More thought will be required if something more sophisticated is needed.

8 Server Environment

8.1 Loading the crate server

In a production environment the server program must be started automatically following a crate power-up or re-boot, whereas when a crate is on the maintenance bench it is unlikely to be of any use (and will probably be an encumbrance). Hence we will need to control what programs are loaded at crate boot-time by means of different boot parameters and scripts—VxWorks experts please advise.

In the case of VXI crates, we anticipate that before the server is started the slot 0 Resource Manager software will have located all occupied slots in the crate, identified the modules, performed some minimal checks (e.g. modules powered up, configuration registers initialised) and assigned the modules VME addresses according to the VXI specification. The RM must provide the server (and other programs) with a "crate map" defining module type, serial number, and VME base address of each occupied slot. The server can check module types and serial numbers against the declarations in the configuration file. The VME addresses provide the server with the *basedat* function. The RM software can either build a suitable globally accessible data structure or export an inquiry function. In the case of non-VXI crates, this information must be fixed by convention (undesirable) or obtained from the configuration description file.

8.2 Diagnostic procedures

This specification makes no mention of diagnostics for modules since we regard this as outside the scope of the crate server program. This is currently a topic under discussion but we anticipate that diagnostics will fall into three classes:

online This class represents tests that can be set up by experimenters by manipulating on-board registers using the facilities of the crate server. These tests would be resorted to first should a problem be suspected while preparing for or running an experiment. They range from simple data integrity tests of individual registers to (possibly) using the Ge card on-board pulse generator to produce a singles histogram. The latter test (if feasible) would exercise readout controller, DT32 bus, and histogrammer functions. At this stage of the design we do not anticipate incorporating extra functions in the crate server to support diagnostic tests. For example, we would exclude anything which required the crate server to field interrupts from modules, this being a job for the next class of diagnostics.

standalone This class of test can be met by running special purpose diagnostic programs in the crate under suspicion. This would require a remote login to the crate and some familiarity with the VxWorks shell in order to load and execute the programs. Such tests could involve only modules within the crate and could not assume any of the functionality of the production data acquisition software. In practice, it may be necessary to displace the standard crate server program before running such a diagnostic and such programs must rely on the crate map built by the Resource Manager at boot-time to identify modules. Tests in this class are candidates for running automatically at boot-time as confidence tests before invoking the crate server program. If written to a local file the results of such confidence tests could be returned by an inquiry function supported by the crate server, analogous to the *ReadEuroGamLog* operation.

maintenance Finally we have a class of diagnostic appropriate to the maintenance bench—cards on extenders, special test harnesses, etc., which we will gratefully abandon to the design/commissioning engineers.

8.3 Server network port

The server will always listen on a fixed UDP port number to be determined.

8.4 Error recovery

Lack of familiarity with the facilities offered by VxWorks prevents us from recommending a specific fatal error recovery mechanism at present. Possibilities which should be investigated include:

- watchdog timers
- remote boot facility

VxWorks experts please advise.

8.5 Scope channels

At this stage of the design it is not clear how the "scope channel" feature of the slot 0 card will work and how it might be used in production. However, we can envisage a set of registers on the card which control the scope function and can be accessed via the crate server. What to do with the data which is produced is an open problem. One possibility is to have a client program polling the crate for the scope data. Another is to provide a pseudo register containing a network address to which the crate calls back and transmits the data.

8.6 Configuration description file

We have shown the configuration description file as an input parameter to the *ConfigureEuroGam* schema. In practice, a client program passes the crate server a file name and the server accesses the file by means of the NFS capability of VxWorks. We expect the description file(s) to reside on some file server system where they are maintained by support staff using ordinary text editors. Temporary changes during an experiment—needed to configure around a hardware fault, for example—are accommodated by modifying a copy of the basic description file. Copies of description files can be taken away to aid post-experiment analysis if need be.

9 Client Programming Issues

9.1 Addressing crates

Client programs must learn the network addresses of crates comprising a EuroGam configuration by reading the appropriate configuration description file, but in general they will not know which crate supports a particular abstract register (though there are exceptions). Solutions to this problem include:

- RPC to all crates (preferably in parallel)—at most one will respond affirmatively. This is the only practical solution in the case of the pattern-matching functions.

- Build a register location map by polling the crates with *InquireRegisters* functions. This may be a viable solution for certain long-running clients.
- Compile the configuration description file—i.e., emulate the crate servers themselves. This is not recommended.

9.2 RPC library

Further work needs to be done in defining a package of routines that generate the above XDR bindings for linking with client programs. This needs to take into account the issues raised in the preceeding paragraph.

9.3 Deadlock avoidance

A classic deadlock can occur if client programs are careless of the order in which they claim crates. This possibility can be avoided if clients only acquire crates in the order in which they appear in the configuration description file. If a client fails to acquire one of the crates comprising a EuroGam configuration it should release any it has succeeded in claiming.

9.4 Detector database

Experimenters have registered some interest in establishing some sort of "detector database." Such a database would be held on a file server system and maintained by the local detector support staff. In particular, users envisage setting up the LeCroy High Voltage supplies by reference to "preferred bias" information held per detector in such a database. This can be realised by means of a client program that interrogates the detector database for the appropriate voltage and invokes the *WriteRegister* function specifying a register name of the form *detector*`.Bias`—the register access mechanism in the appropriate crate server does the rest. The contents and format of such a database are outside the scope of this document. No further applications of a detector database which impact the crate server software have been suggested.

In similar vein, the idea of an "array database" containing, for example, details of the array geometry has been mooted. Again, this topic is outside the scope of the crate server software.

References

[1] *Eurogam Project: System Specification*, Pucknell et al, Edition 1.0, May 1990.

[2] *The Z Notation—A Reference Manual*, J.M.Spivey, ISBN 0–13–983768–X

[3] *Network Programming*, Sun Microsystems, part number 800–1779–10, May 1988.

[4] *Distributed Systems—Concepts and Design*, Coulouris and Dollimore, ISBN 0–201–18059–6