APPENDIX C Readout Controller Configuration File (Language Summary - Version 2.0)

This appendix gives a summary of the data acquisition statements (CODA Readout Language or crl) used to build readout lists for the supported FASTBUS and VME and CAMAC ROC's (readout controllers) as well as the format of the files containing these lists. (Details on the ROCs functions as well as the ROC and readout list architecture can be found elsewhere.) The crl files must be compiled with the *makelist* utility, and the name of the compiled files are passed to the ROC by an entry in a run configuration table of the Run Control database. Finally, the CODA crl->c compiler (*ccrl*) may be used independently of the *makelist* utility to generate resulting c code from the crl file. The readout language is designed to be complete enough for most experimenters, and hides many of the board specific implementation details.

C.1 File Format

Each ROC configuration file is composed of 1 or more sections of code to be executed upon receipt of a corresponding event, either a hardware trigger or a change-of-state command from Run Control. In addition there may be a section of declarations and/or definitions at the top of the file, for example to define constants, global variables, and compiler options. Each section other than the declaration/definition section starts with a "begin section-name" and ends with "end section-name". More experienced programmers can take advantage of the ability to embed their own c code into the crl file directly or via an optional include statement:

```
# comments start with pound signs
readout list Fred    # give list a name
sfi readout          # set compiler for SFI hardware
const SLOTS = 5      # constant definition
variable i           # global variable declaration
include "mycode.h"   # include users c code
begin usercode       # begin section for user specific routine
%%                   /* imbed section of c code using %% */
 ...
%%                   /* c comments also allowed anywhere */
end usercode         # end must have matching name
begin prestart       # hardware initialization
 ...
usercode();          /*single line c code requires semicolon*/
end prestart
```

C.2 Compiler Flags

The first non-comment lines of code select a readout list name, event buffer sizes, what type of readout hardware is being used, and whether the readout will be triggered by interrupt or by polling. The readout list name is specified by

```
readout list SFI1
```

where the name (e.g. SFI1) must be a single word and is case sensitive.

The next statement is optional and specifies the maximum size and number of event buffers to create for the readout list event pool. The default, a maximum event size of 4 Kilobytes and number of 512 buffers (a total of 2 MBytes) is used otherwise.

```
maximum 256,1000          # create 1000 256 Byte buffers
```

If execution of the trigger routine(s) is to be determined by polling of a variable or hardware register then the next statement should be present. This statement is mandatory for *test* and *event* mode readout.

```
polling
```

Next, there must be a line containing one of the key statements for FASTBUS, SFI, CAMAC, or VME to enable support for the corresponding hardware:

```
camac readout             # uses camac standard routines
vme readout               # camac and vme may both be used
fastbus readout           # FSCC fastbus routines included
sfi readout               # SFI/fastbus routines included
test readout              # generates test triggers
event readout             # use for secondary rol only
```

If the fastbus readout option is chosen there are several additional flags which can be set enabling different options. These are,

```
inline fastbus            # Inlines all FB routines for faster
                            execution
nocheck fastbus           # Turns off error checking. Speeds up
                            execution, but should be used only
                            when user is confident that FB
                            readout is operating properly
parallel link             # Redirects data flow through the
                            FSCC output port into a VME memory
                            module. (requires correct hardware)
```

Finally for fastbus, sfi, and vme readout options the following statement can be added to allow external triggers to come from the trigger supervisor.

```
ts control                # Should be used when the FSCC SFI or
                            VME crate is being triggered by the
                            Trigger Supervisor (requires
                            Trigger Supervisor and TS interface
                            cards)
```

C.3 Code Sections

There are 4 types of code sections: a compiler option.global declaration section, readout (trigger) functions, state transition command functions, and user command functions. A minimal but compilable readout list shell (which does nothing except not Crash) looks like as follows:

```
readout list FRED
polling
test readout

begin download
end download

begin prestart
end prestart

begin end
end end

begin pause
end pause

begin go
end go

begin trigger usertrig
end trigger

begin done usertrig
end done

begin done
end done
```

In the above example the rol named FRED is a test list. It contains no global declarations and a begin-end function declaration for each of the Run Control transitions (Download, Prestart, End, Pause, and Go). It contains a single trigger-done pair of routines named ‘usertrig’, and finally it contains the global done routine (unamed).

The CODA Readout software supports the use of multiple trigger sources. For each source there must exist a trigger-done pair that is uniquely named. Readout of an event is complete when all valid trigger sources have been serviced (i.e. trigger routine followed by done executed for that source). Then the global done routine is executed and all sources re-enabled for a new event. If ts control is enabled then the ROC expects the hardware trigger information to be conveyed on the Trigger Supervisor ROC cable connected to a TS interface card (TJNAF design). The ROC receives a 6 bit code (4 trigger type bits, one sync bit, and a late fail bit). This code is generated via memory lookup in the Readout Code MLU of the Trigger Supervisor, with the trigger inputs as an address.

When the ROC receives the trigger from the TS a named “trigger” list will be executed and a local variable called “EVTYPE” will be set with the value of the trigger information from the TS (if ts control is not specified then the default value of EVTYPE=1).

In addition to specifying instructions for hardware triggers, the following state transitions may also have ROC instructions associated with them: *download*, *prestart*, *go*, *pause*, and *end* (with identical readout section names: download, etc.):

```
begin prestart
 ...                    # clear/initialize hardware
end prestart
```

The user may define hist own routines that may be called (or spawned as a task in VxWorks) in any of the transition lists. These routines are created as `void` so no values may be returned:

```
begin userCode
 ...                    # user specific (can be called from
 ...                    # other lists)
end userCode
```

Finally, the user has available a global declaration section for the readout list. Constant definitions, variable declarations as well as include statements may be added to the global section after the final compiler directive but before the first *begin code* statement:

```
readout list FRED
maximum 1024,500
sfi readout

const SLOT = 12          # Constant Slot number of ADC
include “mycode.h”       # include my own C code
unsigned long ped[64];   # declare an array to hold pedistals

begin download
 ...
```

List statements may either be English-like readout statements defined in the next section, or may be any valid c expressions (the file is first passed to a CODA readout language interpreter (ccrl), and then to the c pre-processor and compiler). Individual lines of c code must end with a semicolon. Larger sections of c code may be imbedded by placing %% prior to and at the end of the code section. Very large sections of user c code may best be added by using the include statement in the global section of the readout list.

C.4 Language Elements

CODA Readout Language (crl) statements include flow control, arithmetic operations, and hardware I/O statements. Each of the statements recognized by the CODA pre-processor begins with a keyword, and may have additional keywords or expressions following.

In the statements that follow, optional elements are shown in [], and alternative choices are shown in [] separated by |.

Variables and Expressions

Four byte unsigned integers, with case sensitive names of up to 31 characters, may be declared either at the top of the file (global variable) or within a section (local variable).

```
variable xxx,yyy,zzz
```

Constants may be declared at the top of the file by indicating the keyword *const*, name followed by an equals sign followed by a value:

```
const NSLOTS = 6
```

Expressions may be built up from variables and arithmetic operators: * / + - ( ). Logical expressions may use the conventional logical operators < > == != <= >= or may use English equivalents:

```
is less than
is greater than
is equal to
is not equal to
is less than or equal to
is greater than or equal to
```

Logical expressions may be combined using parentheses and the operators *and, or* or their c equivalents:

```
(xxx is greater than 7) and (yyy is 8)
(xxx > 7) && (yyy == 8)
```

Arithmetic Statements

Constructs exist for clearing, incrementing, and decrementing a variable, as well as assigning an expression to a variable:

```
clear xxx
increment xxx
decrement xxx
xxx = expression
```

Flow Control

There are 4 flow control constructs: *begin...end*, *while...end while, if...else if...else...end if*, and *select on...case...end select:*

```
begin section-name
 statement(s)
end section-name

while logical-expression
 statement(s)
 ...
 break                    # alternative way to exit loop
 ...
end while

if logical-expression then
 statement(s)
else if logical-expression then
```

```
 statement(s)
else
 statement(s)
end if
```

The *else* expressions are optional; there may be as many *else if* clauses as desired.

```
select on expression
case constant1
 statement(s)
case constant2
 statement(s)
...
default
 statement(s)
end select
```

No explicit *break* statement is required in a case clause: flow does not fall from one case into another, but rather terminates at the next case or end statement.

Trigger Elements

For each hardware type (as defined by the compiler keyword *sfi readout*, *camac readout*, etc.) there is the possibility to enable triggers from multiple sources. For instance the SFI fastbus interface can accept triggers from a total of eight external sources (3 NIM inputs, 4 dECL inputs, and the TJNAF TS interface card). Initialization of trigger sources as well as associating a trigger routine with each of these sources should be done in the prestart routine. Subsequent enabling and disabling of the triggers will be handled for the user in the go, pause, and end routines.

First the particular hardware type should be initialized where *hardware_type* = FASTBUS, SFI, VME, CAMAC, TEST, or EVENT:

```
init trig source hardware_type
```

Then the user must link a specific source type to a trigger and done routine:

```
link async trig source h_type s_type to <name> and <name>_done
# or
link sync trig source h_type s_type to <name> and <name>_done
```

The *async* or *sync* keywords indicate a interrupt or polled source respectively. The value of *s_type* is dependent on the hardware_type. A value of 1 for *s_type* is always valid and indicates the TS interface card for SFI, FASTBUS, and VME, a slot=1 LAM for CAMAC, internal triggers for TEST, and the Primary readout list output queue for the EVENT hardware type. Finally, <name> refers to the name of the users trigger routine which should also be defined in the readout list.

Specific physics event types can be enabled for specific source types. Physics event types take on values between 1-15. To enable a type ev_type for a source:

```
event type ev_type then read h_type s_type
```

Event Building

The CODA event format is built upon a bank structure. Typically, one event from a ROC contains a single bank of data of a specific type (i.e. longwords as read from the front

end modules); However, it is often useful to build a ROC event that has a multi-bank structure. The following *open event...close event* and *open bank...close bank* constructs are available to facilitate event building on the ROC:

```
open event type <xxx> of data_type
open bank <xxx> of data_type [code <zzz>]
close bank
close event
```

The variable <xxx> typically contains a unique bank or event type identifier (16 bits). For an event bank this is typically the physics event type (1-15). The *data_type* is a string indicating the type of data that is in the bank or event. For example:

```
BT_UI4                  4 byte integers
BT_UI2                  2 byte integers
BT_CHAR                 null terminated ASCII string
BT_BANK                 a bank containing banks
```

The code variable <zzz> is an optional 8-bit value entered into the low byte of the bank header. For an event bank this defaults to the event number (modulo 256).

Hardware I/O

There are 3 basic hardware operations: read from a hardware module, write to a module, and write to the output data stream. The read from a module also has a variations that allows reading into a variable or directly into the output data stream. In addition, there is a clear crate statement which performs the appropriate operation for that crate.

Output, either explicitly done with the output statement, or implicitly done by a hardware read operation, is assumed to be in units of 4 byte integers. Each time a code section is called, it produces a single bank of 4 byte integers. The bank header (including bank length) is inserted automatically.

Generic I/O

The crate clear or reset operation performs a CAMAC crate clear or Z or a FASTBUS reset. The crate number is ignored for FASTBUS, and defaults to 0 for CAMAC. for the SFI case the crate number corresponds to the base VME address of the SFI as seen from the CPU for A24/D32 address space (i.e. 0xe0e00000).

```
clear crate [number]
reset crate [number]
```

The output statement transfers a single integer variable or constant into the output data stream. The output range statement transfers multiple integers to the data stream from an indexed array:

```
output yyy
output range xx yy Z      # send Z[xx] through Z[yy] into the
                          # output data stream
```

FASTBUS I/O

Reading and writing FASTBUS modules is a 4 step operation: (1) address the module, (2) select which register in the module to read or write (secondary address), (3) transfer 1 or more words (4 byte) of data, and (4) release the module. Modules have a unique

geographical address (slot number, used most often), and may have one or more logical addresses (used in special applications). In addition, they have 2 internal address spaces *data* and *control*. Control space is typically where control registers are found for setting board options, and data space is typically used to read event data.

```
fastbus readout
 ...
geographic data slot-number
geographic control slot-number
logical data laddr
logical control laddr
secondary address saddr
```

The FASTBUS spec allows for addressing of multiple modules at the same time (called broadcast addressing). The syntax for this is

```
broadcast control broadcast_addr
```

where *broadcast_addr* is a module or function specific number. Common examples would be the All Local Module Address 1 or the Sparse Data Scan 9. Refer to specific FASTBUS module manuals on support for broadcast addressing.

Once the addressing has been set up, any number of words may be transferred (depending on the application). The read statement transfers a single word, and the block read statement transfers a variable length block of data (up to an optional *maxwrds* number of data words).

```
read                       # transfer 1 word to output
read into <xxx>            # transfer 1 word to variable
write yyy                  # write yyy to current address
block read [maxwrds]       # transfer block to output
fast block read [maxwrds] # faster block transfer routine
broadcast read into <xxx> # used after a broadcast address
```

Finally, after data transfer the module(s) should be released. The two forms for address/bus release are,

```
release
broadcast release
```

It is often advantageous to execute all four steps in a single fastbus routine call particularly in the user's trigger routines as the time to address-read/write-release is much faster. One may use the following read and write constructs to execute this complete cycle:

```
read from fastbus_addr
read from fastbus_addr into <xxx>
write <xxx> into fastbus_addr
block read [maxwrds] from fastbus_addr
```

For example an address to a fastbus modules control space register 1, a read of the register into a local variable *data*, and subsequent release of the module could be done in the following two ways:

```
geographic control 15
```

```
secondary address 1
read into data
release
# Or in a single fastbus call
read from geographic control 15 secondary address 1 into data
```

CAMAC I/O

CAMAC has a different addressing scheme in which a register in a module is addressed by a combination of branch number (b), crate number (c), slot number (n), internal address (a), and function code (f). The function code generally distinguishes between read, write, and control functions, but may also be used to select between group 0 and group 1 data space (most modules only support group 0). NOTE: The current implementation of CODA only supports branch b=0.

```
camac readout
 ...
read from b,c,n,a,f           # transfer 1 word to output
read from b,c,n,a,f into <xxx>  # transfer 1 word to variable
write yyy into b,c,n,a,f      # write yyy to module
control b,c,n,a,f             # execute control function
```

CAMAC only supports a single trigger (currently) through a CAMAC lam:

VME I/O

There is limited support for addressing and readout of VME modules in crl. Most module access must be done through imbedded c code. However, structures for memory maps of several commonly used modules at CEBAF have been added to aid the user in addressing these modules (See Table 8). These structures are defined when the user specifies `vme readout` at the top of his readout list code. (See the example trigger supervisor readout list in Appendix B.)

TABLE 8

Module	Description	Structure	Pointer
Trigger Supervisor	Control registers	vme_ts	*ts
	Memory	ts_memory[4096]	*tsmem
TS Interface card	Control registers	vme_tir	*tir[2]
Lecroy 1190	Dual ported memory	vme_dpm	*dpm, *dpml
Lecroy 1151	Scaler	vme_scal	*vscal[32]
TJNAF FIFO	Dual ported memory	vme_fifo	“fifo[2]”

The user should be aware of address modifiers and data transfer modes supported by their particular slave modules. The default kernel for the MV162 and MV167 boards used at CEBAF have 4 address spaces defined (Note: this address map is not valid for the PowerPC PPC604 board from Radstone):

```
A16/D16      0xffff0000 - 0xffffffff

A24/D16      0xf0000000 - 0xf0ffffff

A24/D32      0xe0000000 - 0xe0ffffff
```

```
A32/D32       sysMemTop - 0xdfffffff
```

For example the Lecroy 1190 Memory requires certain registers to read and written to with single word transfers (A24/D16) while the memory can be read via longword transfers (A24/D32), hence the definition of two pointers (*dpm, *dpml) which can be defined using the appropriate address modifiers (0xf0xxxxxx and 0xe0xxxxxx).

EVENT I/O

The CODA 2.0 Readout Controller allows a secondary readout list to be downloaded. Its purpose is to accept events from the primary readout list and perform any user specific task such as compression, sparsification, or partial analysis. This type of list is specified by *event readout* keywords at the top of the file. The Event List is always a polling list as it will execute its trigger routine only if there is an event on its input queue (output queue of primary list). The following crl functions are available to Event Lists only:

```
get event                # get an event from input
copy event               # copy event from input to output
                         $ using a Event pool buffer
pass event               # pass an event pointer from input
                         # to output. (fastest)
```

Normally the user would execute the *get event* command first in the Event List trigger routine. This function gets an event from the input queue, and sets the global variables EVTYPE (the event type i.e. 1-15 for Physics) and EVENT_LENGTH (in number of longwords). It also sets up an array INPUT[] in which INPUT[0] points to the first data word of the event. It is left to the user to determine what is to be done with the event. At a minimum a *pass event* or *copy event* must be issued for the event to be passed on to the ROCs output unchanged. As a non-trivial example, the user could open a new event buffer and selectively copy data words from INPUT into the new event buffer:

```
begin trigger eventtrig
variable ii
get event
open event type EVTYPE of BT_UI4
%%
    for (ii=0;ii<EVENT_LENGTH;ii++) {
       if((INPUT[ii]&0xfff) > 300) /*Check if over pedistal*/
          *rol->dabufp++ = INPUT[ii];
    }
%%
close event
end trigger
```

Utility Statements

Arbitrary messages may be sent to the console task, tagged with a severity. This routine should be used with caution inside event readout lists as they may over run the logger's ability to keep up (thereby losing messages and degrading performance). In general messages should NOT be sent inside the user's trigger or done routines. The format of this call is similar to a c printf statement wherein the message string includes embedded format descriptors. For each format descriptor, the next unused argument is fetched and formatted according to the descriptor.

```
log [ inform | warn | alarm ] “quoted string”,args,...
```

For example:

```
log warn “counter value is %d”,counter
```

%d converts a decimal integer, %x produces hex output; other formats may be found in any c manual.

A user may make a function call to any routine defined in the readout list that uses the *begin* <*name*>*...end* <*name*> structure:

```
call <name>               # call the function <name> (no args)
```

C.5 Example File

The following is a listing of the configuration of a readout controller which reads out a single Lecroy 1881 ADC. Triggering is provided by the Trigger supervisor. In the trigger routine, a broadcast address (Sparse Data Scan) is made to determine if the module has valid data in its buffer. The fastbus routines will be inlined providing approximately 50% faster execution of the trigger routine.

```
#Example Fastbus readout code for a single Lecroy 1881 ADC
#SFI is being triggered externally
#David Abbott, TJNAF 1996

readout list ADC1
maximum 512,1000
sfi readout

const SFI_ADDR = 0xe0e00000
const ADCSLOT = 16
const SCANMASK = hex 00010000

begin download
  log inform “User Download”
end download

begin prestart
  variable adcid

  reset crate SFI_ADDR
  init trig source SFI
  link async trig source SFI 1 to mytrig and mytrig_done
  event type 1 then read SFI 1

# Reset, clear ADC
  write hex 40000000 into geographic control ADCSLOT

# Program for no sparsification, Gate from FP
```

```
  geographic control ADCSLOT
  write hex 00000104
  secondary address 1
  write hex 00000080
  release

# READ ADC ID
  geographic control ADCSLOT
  read into adcid
  release
  log inform “ADC ID = %x”, adcid

  log inform “User Prestart Executed”
end prestart

begin end
  log inform “User End Executed”
end end

begin pause
  log inform “User Pause Executed”
end pause

begin go
  log inform “User Go Executed”
end go

begin trigger mytrig
   variable datascan, ii

  open event type EVTYPE of BT_UI4

# loop until ADC is completed buffering

   ii = 0
   datascan = 0
   while ((datascan is not equal to SCANMASK) and (ii < 5))
     read from broadcast control 9 into datascan
     increment ii
   end while

if ii is less than 5 then
# Load next event
  write hex 400 into geographic control ADCSLOT

# Read out ADC
  block read from geographic data ADCSLOT
else
# Output my own header into the data stream
  output hex da0100ff
```

```
  output datascan
end if

close event

end trigger

begin done mytrig
end done

begin done
end done
```