CLAS-NOTE-93-018

TOF Temporary Utility Software

Robert Millner and Elton Smith

Sept. 29, 1993

The Time Of Flight experimental setup requires temporary software that can be used to operate on the data produced by the setup. Specifically, the output of the various devices must be converted from the CODA Raw Physics Event[1] format to the CLAS Event[2] format and CERN-LIB HBOOK Ntupples. The CLAS events may be manipulated with various software being developed by the CLAS software group. The HBOOK Ntupple may be manipulated by PAW or other FORTRAN programs to produce histograms or statistical information on the data. The programs contained herein are temporary in that they were written with the goal of being modifiable to fit our immediate needs. The utilities were derived from programs written by Kevin Beard and John Ficenec (coevb2.exe and Ntup_Coevb2.exe) and libraries written by Kevin Beard (KB_CoEvb). Only a small subset of the utilities written for this project are included herein. These are the "ScDba_map," "CoEvb_unpack" libraries and the "log_ntup" program.

1. See the CODA manual, appendix F for a description of the Physics Raw Event format.
2. See CEBAF CLAS-NOTE-93-002, by L. Dennis and D. P. Heddle for a description of the CLAS format.

CoEvb_unpack library

(by Robert Millner)

Library: /clas71/usr/local/coda/CoEvb_unpack/libCoEvb_unpack.a
Source Dir: /clas71/usr/local/coda/CoEvb_unpack
Additional Libs Required: None
Sources:

CoEvb_unpack_ROC_decode.for
CoEvb_unpack_bank_subs.for
CoEvb_unpack_common_screendump.for
CoEvb_unpack_physics_event.for

Commons:

CoEvb_unpack_physics_event.CMN
(Exportable)

Parameters:

CoEvb_unpack_physics_event.PAR
(Exportable)
CoEvb_unpack_masks.PAR
(Exportable)

Make:

makefile

Comments and Questions to:

Robert Millner
millner@cebaf.gov
millner@vtcc1.cc.vt.edu

The CoEvb_unpack library is a collection of subroutines that work together to strip usable information from the raw physics event file produced by CODA. CoEvb_unpack_physics_event() is a subroutine which takes in an event and unpacks it into common statements using the bank decoder subroutines. CoEvb_unpack_bank_subs.for contains subroutines for decoding particular banks of information. Weather or not to call one of them is determined by bit testing on the second long-word in each bank and checking for a specific result. It is assumed that the incoming bank is a clean event in the CODA standard physics event format. See the CODA manual, appendix F for more information on the specifics of each supported bank type. Since the subroutine only processes banks that it specifically looks for, any other CODA format banks are skipped over. Any standard Physics event should be decodable into usable information in the common statements that another program can access. The file, CoEvb_unpack_ROC_decode.for contains subroutines to actually look at the Readout Controller (ROC) data for information. These are not called directly by the bank subs or the unpack subroutine but are provided as a standard format for taking a raw ROC bank and producing a set of usable data.

Prototypes:

CoEvb_unpack_physics_event.for
(Robert Millner, 7/30/93)

```
integer*8 buffer(*), buffer_size
call CoEvb_unpack_physics_event(buffer,buffer_size)
```

This subroutine will search the array buffer from position 1 to a position passed in buffer_size for banks in a CODA event. To decode a bank, a check is made by applying an "and" bit mask to the second word in the bank and compare to an expected result to determine which subroutine is called. After decoding, the bank decoder sub will advance the pointer to the next bank in the buffer. If the bank cannot be decoded then it will be skipped. The subroutine is selective about when it can receive a certain kind of bank. It will not interpret a "prestart" event if it has already received a normal physics event header on that event. It will only interpret a "go" event if it has received a "prestart" before in that run. An "end" will be interpreted only if it has already received a prestart and go and has not received an event header. A event header can be interpreted only once each event. ID and Readout Controller (ROC) banks will be decoded only if an event header has been seen on that event, in addition, ROC banks will only be decoded if an ID event has been received first. After a call to this subroutine, data from the events may be read from the common blocks. Banks that are dependant on other banks being called first may have their com-

mon block information reset. For instance, a "prestart" will reset all of the banks. A physics header will reset the ID and ROC commons.

CoEvb_unpack_bank_subs.for
(Robert Millner, 7/30/93)

```
integer*8 buffer(*), point
```

(Buffer is the data space for the bank to be read, point is the location of the start of the bank in that buffer.)

```
call CoEvb_unpack_unload_header(point,buffer)
call CoEvb_unpack_unload_id(point,buffer)
call CoEvb_unpack_unload_event_roc(point,buffer)
call CoEvb_unpack_unload_prestart(point,buffer)
call CoEvb_unpack_unload_go(point,buffer)
call CoEvb_unpack_unload_end(point,buffer)
```

Each of the "unload" subroutines will take their appropriate bank from the CODA Physics event in array "buffer" starting at the location passed in "point" and copy pertinent information into the library common blocks. This information may then be used by any other program linking to the common blocks. At the end of the call, point is set to the position immediately after the last long-word in the bank.

In order to add more bank types, the user must first determine an AND bit mask against the second longword that will produce a certain result and include that as an "if" test in the CoEvb_unpack_physics_event main loop. That information should be placed in the parameter file. Any suitable conditions for ignoring it anyway must then be determined and included in the "if" statement. A subroutine should be written following the standard template that was used in creating the included subroutines. Information produced for the rest of the program should be passed in the common blocks. The subroutines were written this way in order to facilitate modifying or adding components.

CoEvb_unpack_ROC_decode.for
(Robert Millner 7/30/93)

```
integer*8 place,slot(*),channel(*),value(*)
call CoEvb_unpack_roc0(place,slot,channel,value)
call CoEvb_unpack_roc1(place,slot,channel,value)
```

Each of these subroutines is designed to take the information for the appropriate Readout Controller (ROC) in the common blocks, decode it into usable parameters, and place it into arrays of the slot id of the card, channel on the card and value in the channel. These are not called by the rest of the unpack subroutines as they are not needed to unpack a CODA Physics event itself, but rather the device specific information contained therein. Subroutine CoEvb_unpack_roc1() takes all data coming in from CAMAC on Readout Controller 1 and places it into the value arrays. Subroutine CoEvb_unpack_roc0() strips out the information from either a Phillips 10C6 TDC, a Lecroy 1872A TDC or a Lecroy 1881 ADC on FASTBUS (Readout Controller 0). Information is decoded in the order that it was placed. These subs are basically just working prototypes of one possible way of decoding an event.

CoEvb_unpack_physics_event.PAR

(Robert Millner 7/30/93)

This file is the exportable parameter information that may be useful to define arrays. The variable evroc_max_roc_num controls how many ROC's to allocates space for. The variable evroc_mac_roc_data controls how many places in the array are allocated for each ROC. These numbers are set low until higher values are needed.

integer*8 CoEvb_unpack_evroc_max_roc_num

integer*8 CoEvb_unpack_evroc_mac_roc_data

CoEvb_unpack_physics_event.CMN

(Robert Millner 7/30/93)

These variables are used to pass information from the CODA Physics event to other subroutines or a program. To access the information produced by the unpack_event and bank_subs directly, the program must be linked to these commons. See the CODA Manual, appendix F for more information on the CODA Physics event information. Some commons will retain their information for an entire run and some bank subroutines will modify the common information for other commons than its own. Specifically, a HEADER bank will reset the ID and ROC bank. A PRESTART bank will reset the entire commons and GO events will reset END events. Unless reset, PRESTART, GO and END events will retain information throughout the entire run. HEADER, ID and ROC events are reset each event.

Event Header bank def.

```
integer CoEvb_unpack_evhead_event_end,
        CoEvb_unpack_evhead_event_type,
        CoEvb_unpack_evhead_event_len
logical CoEvb_unpack_evhead_got
```

```
common /CoEvb_unpack_evhead/
        CoEvb_unpack_evhead_event_len,
        CoEvb_unpack_evhead_event_end,
        CoEvb_unpack_evhead_event_type,
        CoEvb_unpack_evhead_got


                        Event id bank def.
integer CoEvb_unpack_evid_event_clas,
        CoEvb_unpack_evid_event_num,
        CoEvb_unpack_evid_status_sum
logical CoEvb_unpack_evid_got

common /CoEvb_unpack_evid/
        CoEvb_unpack_evid_event_clas,
        CoEvb_unpack_evid_event_num,
        CoEvb_unpack_evid_status_sum,
        CoEvb_unpack_evid_got


                        ROC bank
integer CoEvb_unpack_evroc_roc_data_len
                (0:CoEvb_unpack_evroc_max_roc_num)
integer CoEvb_unpack_evroc_roc_data
                (0:CoEvb_unpack_evroc_max_roc_num,
                1:CoEvb_unpack_evroc_mac_roc_data)
logical CoEvb_unpack_evroc_got
                (0:CoEvb_unpack_evroc_max_roc_num)

common /CoEvb_unpack_evroc/
        CoEvb_unpack_evroc_roc_data_len,
        CoEvb_unpack_evroc_roc_data,
        CoEvb_unpack_evroc_got
```

Run SYNC

(Ignore the sync events for now.)

Run PreStart

```
integer CoEvb_unpack_runps_time,
        CoEvb_unpack_runps_run_num,
        CoEvb_unpack_runps_run_type
logical CoEvb_unpack_runps_got

common /CoEvb_unpack_runps/
        CoEvb_unpack_runps_time,
        CoEvb_unpack_runps_run_num,
        CoEvb_unpack_runps_run_type,
        CoEvb_unpack_runps_got
```

Run Go

```
logical CoEvb_unpack_rungo_got

common /CoEvb_unpack_rungo/
        CoEvb_unpack_rungo_got
```

Run Pause

(Ignore Pause)

Run End

```
integer CoEvb_unpack_runend_time,
        CoEvb_unpack_runend_num_events
logical CoEvb_unpack_runend_got

common /CoEvb_unpack_runend/
        CoEvb_unpack_runend_time,
        CoEvb_unpack_runend_num_events,
```

CoEvb_unpack_runend_got

CoEvb_unpack_common_screen_dump.for

(Robert Millner 7/30/93)
call CoEvb_unpack_common_screen_dump()

This subroutine will test weather or not a particular type of bank has been loaded into the commons and print the commons of each type it could find to the standard IO.

ScDba_map library

(by Robert Millner)

Library: /clas71/usr/local/coda/ScDba_map/lib_Sc_Dba_map.a

Source Directory: /clas71/usr/local/coda/ScDba_map

Additional Libs Required: None

Sources:

ScDba_map_init.for

ScDba_map_logical.for

ScDba_map_logical_event.for

ScDba_map_physical.for

ScDba_map_physical_event.for

ScDba_map_tester.for

Commons:

ScDba_map.CMN

(Not Exportable)

Parameters:

ScDba_map.PAR

(Not Exportable)

Make:

makefile

Comments and Questions to:

Robert Millner

millner@cebaf.gov

millnerrl@vtcc1.cc.vt.edu

Overall Description

The ScDba_map library is a collection of subroutines that work together to map the physical[1] parameters of the readout controllers in a CODA RAW Physics event to and from the logical[2] parameters of a CLAS Packed event. ScDba_map_init() must be called first when accessing the library in order to load the map into memory and make the necessary initialization. ScDba_map_logical() and ScDba_map_logical_event() translate a single element and an entire event from its physical parameters to the CLAS logical arrangement. ScDba_map_physical and ScDba_map_physical_event translate a single element and an entire event from CLAS logical parameters to its physical parameters. The common and parameter blocks are not designed to be exportable and should not be linked into any external programs or libraries as they do not follow the CLAS software standards.

Subroutine Prototypes and Descriptions

ScDba_map_init(filename,ok,error)

Input Variables:

character*(*) filename {The name of the Map file}

Output Variables:

character*(*) error {A text message of the error}

logical ok {T or F, success of the operation.}

This subroutine must be called first when accessing this library. It will read a file containing the translation of CODA physical parameters to the CLAS logical parameters and place it into arrays in the common blocks. Each translation in the file is a line of eight integers for the physical and logical parameters in the order "crate slot channel sector package sub-package element parameter." This provides an exact map on each line. The lines of the file need not be in any kind of order and placing them in one will not optimize their usage significantly. The rest of the subroutines in this library will return an error if they are called before this subroutine.

ScDba_map_logical(crate, slot, channel, sector, package, subpakg, elem_id, parameter, ok, error)

Input Variables:

integer crate {The crate ID of the element being translated}

integer slot {The slot ID.}

integer channel {The channel number of the element being translated.}

Output Variables:

1. Crate ID, Slot ID and Channel Number
2. Sector ID, Package ID, Sub Package ID, Element ID and Parameter ID

integer sector {Assigned sector address.}

integer package {Assigned package number.}

integer subpakg {Assigned sub package number.}

integer elem_id {Assigned element ID.}

integer parameter {Assigned parameter ID of the element.}

logical ok {True or False report of a good translation operation.}

character *(*) error {A text error message.}

This subroutine will accept the physical address of the element being translated in the input variables and produce the CLAS logical address of the element in the output variables. Ok is set to true if the mapping was completely successful. Any parameters that could not be successfully mapped return a -1 as a value. ScDba_map_init() must be called before this subroutine in order to initialize the map.

ScDba_map_logical_event(n_element, crate, slot, channel, sector, package, subpakg, elem_id, parameter, found, ok, error)

Input Variables:

integer n_element(*) {The total number of elements to be mapped in this event.}

integer crate(*) {An array of the crate IDs of the elements being translated.}

integer slot(*) {An array of the slot IDs.}

integer channel(*) {An array of the channel numbers being translated.}

Output Variables:

integer sector(*) {An array of assigned sector address'.}

integer package(*) {An array of assigned package numbers.}

integer subpakg(*) {Assigned sub package numbers.}

integer elem_id(*) {Assigned element IDs.}

integer parameter(*) {Assigned parameter ID of the elements.}

logical found(*) {True or False for each element which mapped properly.}

logical ok {True or False report of the entire translation operation.}

character *(*) error {A text error message.}

This subroutine maps the physical address' of an entire event to logical parameters. It will start with the first element in the arrays and translate 'n_element' elements by calling ScDba_map_logical() on each element. The variable found returns the status of each operation while ok returns the status of the entire operation.

ScDba_map_physical(sector, package, subpakg, elem_id, param, crate, slot, channel, ok, error)

Input Variables:

integer sector {Logical sector ID of the element.}

integer package {Logical package ID of the element.}

integer subpakg {Sub package ID.}

integer elem_id {Element ID}

integer param {Parameter ID of the element.}

Output Variables:

integer crate {Physical Crate ID of the element.}

integer slot {Slot ID of the element.}

integer channel {Channel ID of the element.}

logical ok {True or False status of the operation.}

character*(*) error {Text error message.}

This subroutine maps an element in the CLAS Packed format to its physical address. Weather or not this was completely successful is returned in the variable ok. Missing information or an erroneous search will return a value of -1. ScDba_map_init must be called before this subroutine to initialize the map.

ScDba_map_physical_event(n_elements, sector, package, subpakg, elem_id, param, crate, slot, channel, found, ok, error)

Input Variables:

integer n_elements {Number of elements in the event to be mapped.}

integer sector(*) {Logical sector IDs of the event.}

integer package(*) {Logical package IDs of the event.}

integer subpakg(*) {Sub package IDs.}

integer elem_id(*) {Element IDs.}

integer param(*) {Parameter IDs of the event.}

Output Variables:

integer crate(*) {Physical Crate IDs of the event.}

integer slot(*) {Slot IDs of the event.}

integer channel(*) {Channel IDs of the event.}

logical found(*) {Mapping status of each element.}

logical ok {True or False status of the operation.}

character*(*) error {Text error message.}

This subroutine maps an entire event of CLAS logical parameters to its physical locations. It will start at the first position in the arrays and map 'n_element" positions by calling ScDba_map_physical(). The array "found" returns the status of each mapping operation in the event. The variable "ok" returns the status of mapping the entire event. ScDba_map_init must be called prior to calling this subroutine.

log_ntup.exe
(by Robert Millner 9/15/93)

This program fulfills two purposes: first to be a testbed for the CoEvb_unpack library, second to be a useful utility for converting CODA logfiles from one 64 channel device into an HBOOK Ntupple. The program will prompt for the filenames of both the CODA log and HBOOK rzdat files to be used. The structure of the output ntupple is outlined in table 1. Data from the logfile will be inserted if it has either or both of CAMAC info on ROC1 and FASTBUS info on ROC0. The program uses the CoEvb_unpack library to decode the CODA Physics Raw event file; therefore, anything that package can decode will be inserted. This program will not work with multiple devices on the fastbus. If it is fed information from more than one device, then whatever had that channel last will be loaded in. If any information is missing, that field will default to zero.

Table 1:

ID	TAG	DESCRIPTION	Type
1	0	FASTBUS Channel 0 value	REAL
2	1	FASTBUS Channel 1 value	REAL
N	N-1	FASTBUS Channel (ID-1) value	REAL
64	63	FASTBUS Channel 63 value	REAL
65	qtg	CAMAC Phillips 7120 Charge Time Generator value	REAL

The body of this program actually lies in two subroutines and the library. The first subroutine takes care of all aspects of loading in the log file. Optionally, CAMAC data from the RunCamac.exe program may be loaded in from a specified file. If no filename is specified, then this option is ignored. The second subroutine takes care of all aspects of booking the ntupple and filling the rzdat file. Both subroutines open all necessary files and close them depending on their input values. Both also return error checking. If the program cannot open the appropriate files, then it will terminate with an error message. At the time of writing, there is a bug in one of the libraries that causes the program to segment fault when it tries to close the log file. This is done last so it does not interfere with the operation of the rest of the program.

The executable, sources, makefile and docs for this program can be found in /clas71/usr/local/coda/Log_Ntup. If a recompile is necessary, then the libraries evgen, evfio (both by Larry Dennis), _KBB_tricks, _ULTRIX_special (both by Kevin Beard), CoEvb_unpack (by Robert Millner) and CERN packlib must be accessible. Beware when recompiling that a newer version of CERNLIB may be present than in other software making the output ntupples unaccessible.

The subroutines load_physics_event.for and test_ntuple.for were intended to be modified until useful and then placed into a library. As is, they are functional; however, test_ntuple may need to be modified to keep up with different types of ntupples needed. The filling is done by looping through the input arrays until either all the tags are filled or the end of the input array is reached. Load_physics_event works with whatever is given to it that the CoEvb_unpack library can decode. If the library is unable to access a piece of information in the log file, it is ignored.

Program termination occurs when a RunControl run-end event is reached in the file. The prestart and go event are needed to allow the unpack library to see the end event in a file; however, they are not used for the rest of this program. This dependency can be taken out of the library if necessary.

Questions or comments to Robert Millner, millner@cebaf.gov.