TAS Applications Manual

Version 2.0

D. L. Olson

19 February, 1993

Contents

Contents....1
1. Introduction....2
    1.1. The TAS data processing model....2
    1.2. TAS data structures: tables....2
    1.3. Analysis modules....3
    1.4. Filters....4
    1.5. Plots....4
2. User's Guide....5
    2.1. User interface, KUIP....5
    2.2. TAS tables and variables....5
        2.2.1. set/show variable commands....6
    2.3. Plot definitions and filling....6
        2.3.1. plot commands....6
        2.3.2. row selections....7
        2.3.3. plot filters....8
        2.3.4. more on ntuples....8
    2.4. Table filters....10
        2.4.1. commands....10
        2.4.2. miscellaneous comments on filters....11
        2.4.3. when expressions....11
    2.5. User function status codes....11
    2.6. Analysis modules....12
    2.7. Data I/O....12
    2.8. Calibration file procedures....13
3. Programming Guide....15
    3.1. tas_gen interface generator....15
    3.2. Style guidelines....16

3.3. Analysis modules 16
3.3.2. TXM_AMSAMP 16
3.4. TAE_USER(mode,tptrs,ntables) 18
3.5. Use of tables 19
4. Functional description of TAS 20
4.1. Architecture and future extensions 20
4.2. Event processing, or what happens when you type GO 23
Appendices 25
A. Data Structure Management System (DSM) 25
A.1 Table editor 26
A.1.1 editing fields 28
A.1.2 example 28
A.2 Code generation 30
Index 31

1. Introduction

The TPC analysis shell (TAS) is a general purpose analysis program for event-based data processing. It is designed to handle many of the difficulties arising when doing analysis in a group environment. The major features contributing to this are:

- a specification for connecting analysis routines
- a separation of the I/O and control functions from the analysis routines
- a single point of input for specifying the data structures
- data structures which are self initializing and self documenting
- interfaced with a widely accepted graphics and interactive analysis package[1] (PAW)

Having a specification for connecting analysis routines and separating the I/O and control functions from these routines permits people to independently develop separate stages of the analysis and then combine the results into a single program without undue difficulty. The single-point data-structure definition is based upon specifying the TAS data structures in a commercial database and generating the necessary source code and include files from this database. The variable names, comments and initial values specified in the database are then carried on through the source code automatically. The variables names specified initially are the same names used for the interactive analysis (PAW ntuple tag name = TAS variable name). By using the widely accepted Physics Analysis Workstation (PAW) package from CERN the user interface and most of the interaction is already familiar to many people.

The remainder of this section gives descriptions of the concepts of data processing, tables, analysis modules, filters and plots as used in TAS. The following sections contain a user's guide (2), a programming guide (3), a functional description of TAS (4), and a description of the data-structure specification method using INFORMIX (A).

1.1. The TAS data processing model

The philosophy of data processing in TAS is as follows. Data is processed as a sequence of steps from raw data into physics summary data. The scope of the data available to the analysis routines at any given time is a single event.

The data format between each step of processing is TAS tables. The complete description of the event at any stage of processing will typically consist of several tables.

Each step of processing is performed by an analysis module. Each analysis module will use either raw data or one or more tables as input and generate one or more tables as output. In this way, an analysis module can be seen as a subroutine that performs an operation on a set of tables.

A set of tables can be saved into an event stream data file at any stage of processing. This can be read in later as the input event stream and processing will resume at that stage. This permits data processing to be performed either as a single pass from raw data to physics summary data (after developing all of the analysis routines, of course) or as a series of passes while saving the data at each stage.

1.2. TAS data structures: tables

The data structures used by TAS are called tables. They are essentially the same as relational database tables or the Aleph Data Model (ADAMO) tables. A table is a 2-dimensional object with columns and rows. Columns correspond to individual variables and rows correspond to multiple instances of these sets of

---

[1] PAW - Physics Analysis Workstation, CERN-DD, Q121, 1989.

variables. TAS tables are implemented as Fortran STRUCTURES[2] (and as C structs). Each row is one RECORD (struct) with a list of variables and the table is an array of these row RECORDS (structs). A table contains data for one event and a counter (called NOK for number of OK rows) indicates the number of rows with valid data. For example, a table for a Time-of-Flight wall looks like

TOF table with 4 slats hit.

Variables / Values								
data types	I*4	I*4	I*4	I*4	I*4	R*4	R*4	R*4
row	slat	tdc_t	tdc_b	adc_t	adc_b	deltat	tbar	abar
1	1	213	392	246	283	-89.5	302.5	263.85
2	3	1123	1046	554	456	38.5	1084.5	502.62
3	4	1712	1643	53	-	34.5	1677.5	0
4	23	-	1987	147	183	-993.5	993.5	164.02

row = row index in table, slat = slat number, tdc_t = top pmt TDC,
tdc_b = bottom pmt TDC, adc_t = top pmt ADC, adc_b = bottom pmt ADC,
deltat = (tdc_t - tdc_b) / 2, tbar = (tdc_t + tdc_b) / 2, abar = $\sqrt{adc_t \bullet adc_b}$

An example for a parts (particle source) table looks like

Parts table with 4 particles.

Variables / Values					
data types	I*4	R*4 p4(4)	I*4	I*4	R*4
row	ip	p4	ia	iz	mass
1	1	(.1,.23,.765,.817)	0	-1	0.139
2	2	(-.34,.05,1.34,1.67)	1	1	.938
3	3	(.32,-.25,1.98,2.758)	2	1	1.876
4	4	(-.45,.51,3.67,5.275)	4	2	3.727

row = row index in table, ip = particle number,
p4 = 4-momentum (vector length 4), ia = atomic mass number,
iz = charge, mass = mass of particle

1.3. Analysis modules

Specific analysis routines connect to TAS in a form called analysis modules. Each analysis module accesses certain tables in a read, write or update mode. The mode of access to these tables is declared to TAS in the initialization phase of running the program. The table data structures are passed to the analysis module as

---

[2] TAS V2.0. See your favorite Fortran manual under VAX extensions.

arguments of the top level subroutine of the module. See the programming guide (pg. 16) for an example of this.

The TAS shell provides tools to manipulate tables; read and write to files, fill plots, make cuts, etc., so that analysis modules can be kept simple and free of complicated control features. Program development will be aided by keeping in mind the concept that an analysis module is simply an operator that generates new tables from existing tables.

1.4. Filters

TAS has two kinds of filters, plot filters and table filters. Both of these are implemented as calls to COMIS[3] functions which can be either compiled or interpreted Fortran routines.

A plot filter is applied to the rows of a table which are being used to fill a plot (1-D, 2-D histograms or ntuples). The values from each row of the table are used to fill the plot based upon whether or not it passes the filter. Plot filters do not delete rows from tables but merely affect plot filling.

Table filters are applied to the rows of a table at a specified point in time; before, in between, or after calls to analysis modules. These filters actually delete rows from the table based upon the filter function.

A more complete description of filters is given in the User's Guide (pgs. 9, 11).

1.5. Plots

TAS uses the CERN PAW program for graphics and interactive analysis. In addition to the normal PAW commands, TAS provides an interactive means specifying how to fill HBOOK 1-D and 2-D histograms and ntuples from TAS tables. The User's Guide section (pg. 7) provides a description of defining and filling plots in TAS.

---

[3] COMIS - Compilation and Interpretation System, User's Guide, CERN, L210, 1988. A Fortran interpreter, see also the PAW manual.

2. User's Guide

The User's Guide for TAS is primarily the on-line manual available from KUIP[4] which is the user interface package for PAW and TAS. This section provides some amplification on various features of using TAS interactively.

2.1. User interface, KUIP

The user interface of TAS is the KUIP package used by PAW so anyone familiar with PAW will already know many of the features. I will just mention some of them here and leave further detail to the PAW Manual.

KUIP has both brief (`usage`) and verbose (`help`) on-line help. The `usage` command shows the specified command line format indicating which arguments are optional or required. The `help` command shows the command line format, a short description of each of the arguments and a long description of the command.

KUIP has various styles of interface; command line, alphanumeric menu or graphics menu modes. A description of these are given in both the PAW and KUIP manuals.

KUIP allows for command name abbreviation so that the shortest unambiguous name for a command is permitted.

By default all commands are recorded in a file called `last.kumac` in your current working directory. These command lists can be copied and edited to produce macro files which can be executed by KUIP. This macro facility in KUIP is programmable including local and global[5] variables, looping, conditional branching, operators, and subroutines.

In addition to the `pawlogon.kumac` file that PAW normally tries to execute at program startup, TAS will execute a KUIP macro file called `taslogon.kumac`, if it exists. This file can be used to store settings, definitions, etc., that get loaded automatically when TAS is executed.

2.2. TAS tables and variables

From the command line in TAS variables are referenced as *table*[(*rowsel*)].*variable*[(*elem*)] where *table* is the name of the TAS table, *variable* is the name of the variable, (*rowsel*) is an optional specification of a selection of rows in the table and (*elem*) is an optional specification for elements of a vector variable. The row (element) selection is one of the following forms

1. *nothing* - no row (element) selection means select all rows (elements),
   Ex: tof.adc_t will select all rows in the tof table for the variable adc_t.
   (parts.p4 will specify elements 1 through 4 of the vector variable p4.)
2. (*n*) - selects row (element) *n*
   Ex: tof(3).adc_t selects row 3 in the tof table for variable adc_t.
   (parts.p4(4) specifies element 4 of the vector variable p4.)
3. (*n*:*m*) - selects rows (elements) *n* through *m* inclusive
   Ex: tof(2:5).adc_t selects rows 2 through 5 in the tof table for the variable adc_t.
   (parts.p4(2:3) specifies elements 2 and 3 of the vector variable p4.)
4. (:*m*) - selects rows (elements) 1 through *m* inclusive.
5. (*n*:) - selects rows (elements) from *n* to the last row (element).

---

[4] KUIP - Kit for an User Interface Package, CERN-DD, I202, 1988.

[5] KUIP vectors act as global variables.

6. (*c*) - a special form of row selection used in defining how plots are filled. *c* is an arbitrary character string. See the section on plot filling below.
   (This form is not applicable to the elements of vector variables.)

Any particular variations or restrictions to the above rules are listed in the help for those commands.

TAS variables can be either scalars or vectors (except that a vector of length one is not allowed) but no higher dimension arrays. Scalar variables are referenced by name with wild cards allowed, i.e., `tof.a*` will select the variables `adc_t` , `adc_b`, and `abar` for all rows of the tof table.

The data types allowed are INTEGER*4, REAL*4 and CHARACTER*4N (N is the dimension of the character variable specified with the TAS table editor). This means that some data formats can not take advantage of the built in TAS features. Since most raw data is 2-byte format and not 4-byte format one cannot fill plots directly from an INTEGER*4 variable and get sensible results. Additional data types like INTEGER*2, LOGICAL*1, REAL*8 will probably be implemented in the near future. However, note that these additional types, which are not native to the COMIS interpreter, will require special treatment when passed to COMIS filter functions.

2.2.1. set/show variable commands

Three of the TAS commands provide interactive access to table variables. These are for showing the definition of the variables and also for setting and showing the current values of the variables.

- SHOW/TABLES [tabvar]
  will list the tables and variables which are defined
- SET/VALUE tabvar value
  will set the variables specified by tabvar to the specified value
- SHOW/VALUE tabvar [unit prefix]
  will list the values of the variables specified with tabvar to Fortran logical unit (unit=6 by default) with an optional prefix string (blank by default). With prefix = 'set/value' one can make a command file to save the present values which can be read later to restore these values (via the SET/VALUE command).

2.3. Plot definitions and filling

TAS provides a means of defining and filling HBOOK 1-D, 2-D histograms and ntuples from any variables in TAS tables. Variables are specified on the command line as described in the previous section. The TAS commands for creating, deleting, defining fill variables and filters must be used consistently for histograms and ntuples known to TAS. One should not create a histogram with TAS commands and then delete it with the PAW command HIST/DELETE since TAS will still try to fill a histogram which has been deleted from HBOOK.

2.3.1. plot commands

A brief description of the command sequences for manipulating TAS plots is given here. The long description of each command is given in the on-line manual.

- **1-D histograms**
  DEFINE/1DHIST/NEW idh title nx xmin xmax [valmax chopt]
  Creates (books) a 1-D histogram with the specified parameters.
  DEFINE/1DHIST/VARIABLE idh xvar [weight]
  Defines which TAS variable **xvar** (a t**able.variable** expression) is used to fill the specified histogram. An optional **weight** variable may be specified (also a **table.variable** expression), by default a weight of 1 is used. Issuing this command again will assign a new variable to the existing histogram and the contents of the histogram are left intact.
  DEFINE/1DHIST/FILTER idh cfunc
  Specifies a filter function for the histogram.

DELETE/PLOT idlo [idhi]
Deletes the specified plot or plots (all plot types).
SHOW/PLOTDEF [idlo idhi]
Lists the definitions for the specified plots (all plot types).

- **2-D histograms**
  DEFINE/2DHIST/NEW idh title nx xmin xmax ny ymin ymax [valmax chopt]
  Creates (books) a 2-D histogram with the specified parameters.
  DEFINE/2DHIST/VARIABLES idh xvar yvar [weight]
  Defines which TAS variables are used for filling the specified histogram. **xvar** , **yvar** and **weight** are **table.variable** expressions. **weight** is optional and a weight of 1 is used by default. Issuing this command again will assign new variables to the existing histogram and the contents of the histogram are left intact.
  DEFINE/2DHIST/FILTER idh cfunc
  Specifies a filter function for the histogram.

- **ntuples**
  DEFINE/NTUPLE/NEW idn title [chrzpa]
  Starts a ntuple definition with the specified parameters.
  DEFINE/NTUPLE/VARIABLE tab_var [tag vnum idn]
  Assigns variables to be used for the ntuple. This command can be issued many times to define up to the maximum number of variables allowed for an ntuple[6].
  DEFINE/NTUPLE/BOOKIT [idn]
  Closes the definition command sequence and books the specified ntuple. This command can be reissued and it will delete the existing ntuple and book a new one with the definition specified for **idn** . This is roughly equivalent to the PAW reset command for histograms.
  DEFINE/NTUPLE/FILTER idn cfunc
  Specifies a filter function to be used when filling the ntuple **idn** .

For 1-D and 2-D histograms it is necessary to issue both the .../NEW and .../VARIABLE commands to complete a TAS definition for histograms. The necessary command sequence for a TAS ntuple definition is NT/NEW followed by at least one (but possibly many) NT/VARIABLE commands and terminated by DEFINE/NTUPLE/BOOKIT. For all plots the .../FILTER command may be given any time after defining the plots and the filter will affect plot filling from that point on.

2.3.2. row selections

The row selections mentioned in the previous section are central to the mechanism of plot filling in TAS. This permits everything from filling a histogram based upon a single variable in one row of a table to filling histograms and ntuples based upon many variables from many tables with a complex specification for how to combine rows from different tables.

The basic concept is that a **table.variable** expression selects a set of rows for a given variable (or variables if wild cards are used). If no explicit row selection is specified (i.e., tof.adc_t and not tof(2:4).adc_t) the implied selection is all valid rows of table and the row selection expression is just the table name. In a case where more than one row selection is specified in the set of table.variable expressions used to define a given plot, the situation arises that one must consider how to combine the rows from the different tables, or different selections from the same table. All variables selected with the same row selection expression are combined on a one-to-one basis with equal row index. Variables selected with different row selection expressions are taken from all possible combinations of rows.

---

[6] This limit is 512 variables in PAW 1.13.

For example, consider a 2-D histogram with variables specified as
2D/VAR 10 tof(1:3).abar tof(3:5).abar
This will fill the histogram 10 with 9 entries[7] for each event, 3 x values (1:3) times 3 y values (3:5).

As a more realistic example, consider a 2 layer time of flight wall where one wants to correlate the pulse height in the front and back slats in a 2-D histogram. The histogram variables and filter are specified as
2D/VAR 10 tof(*front*).abar tof(*back*).abar
2D/FILT 10 isfb.for(tof(*front*).slat,tof(*back*).slat)
where arbitrary character expressions are used for the row selections and a filter function is included to narrow the range of combinations plotted. The keywords *front* and *back* are used to correlate the x and y histogram variables with the arguments used in calling the isfb.for filter function. This will yield the same result as the following Fortran code segment:

```
DO i = 1, NOK
    DO j = 1, NOK
        IF ( isfb( tof(i).slat, tof(j).slat ) .gt. 0 ) THEN
            CALL HF2( 10, tof(i).abar, tof(j).abar, 1. )
        END IF
    END DO
END DO
```

where NOK is the number of valid rows in the tof table and HF2 is the HBOOK routine that fills a 2-D histogram. The COMIS function isfb.for would presumably return a value of 0 except where the arguments are slat number combinations that correspond to adjacent front and back slats.

For each plot, the set of row selection expressions used is taken from all variable, weight and filter function expressions. At the present time TAS permits up to 5 different row selection expressions for each plot. Since this method of joining different tables together in plots is not very clever, you should take some care since one can define plots that take a very long time to fill.

2.3.3. plot filters

Plots are filled after calling all analysis modules for each event. The definition of a plot specifies a selection of rows or combinations of rows from tables. The variables that go into filling a plot are taken from these rows. The plot filter function is called for each row (or combination of rows) before filling the plot. If the return value of the function is greater than zero then the plot is filled for that row. If the return value of the function is zero or negative then the plot is not filled for that row and the next row is considered. Note that the return values that apply to other user routines (analysis modules, table filters and TAE_USER) that can stop event processing have no effect for plot filters other than preventing plot filling for that row.

The arguments given to the filter function are copies of the actual variables in TAS so that a plot filter cannot change a value in a table. The data types of the arguments are the same as those for the requested TAS variables so care must be taken that the Fortran code of the filter function uses the same data types.

2.3.4. more on ntuples

Ntuples are specified according to the commands listed above. The effects of some of the command parameters that need more clarification are discussed here.

Since it takes several commands to specify an ntuple, TAS uses the concept of a **current** ntuple definition. The NTUPLE/NEW command begins a **current** ntuple definition. This remains active until the next NTUPLE/BOOKIT command. Any NTUPLE/VARIABLE commands given while a **current** ntuple definition is active will use the **current** ntuple ID (idn) as the default.

---

[7] Assuming at least 5 rows are valid in the tof table.

The **chrzpa** parameter in the NTUPLE/NEW command is the same as in the PAW NTUPLE/CREATE command. It is the RZ directory path name specifying where, in a particular RZ file, the ntuple extensions will be written. When ntuples are filled the entries are stored in the /PAWC/ common block. If **chrzpa** = ' ' then the ntuple filling continues until the /PAWC/ common block is full and the ZEBRA memory manager for HBOOK crashes and TAS crashes. If you have opened or created an RZ file with the PAW command HIST/FILE 30 ... (opens an RZ file attached to Fortran logical unit 30) then you can create a new TAS ntuple with the command
NTUPLE/NEW 10 my_ntuple //LUN30
where **chrzpa** = //LUN30, the HBOOK ntuple ID is 10, and the title is my_ntuple. This ntuple gets filled in /PAWC/ until the initial buffer of 10,000 words is filled, this is then copied to the RZ file (//LUN30), the space in /PAWC/ is cleared and filling continues. When you are finished filling this ntuple you then issue the PAW commands CDIR //LUN30 and HROUT 10. This flushes the remainder of the ntuple into the RZ file and the entire set of entries for that ntuple can then be accessed from that file. This permits arbitrarily large ntuples without the restriction of the size of the /PAWC/ common block.

This is further illustrated by the command sequence below which shows an example starting with creating the histogram file and continuing through flushing the ntuple before closing the file. Note that the ZEBRA/RZ/MAKE command allows one to create larger capacity histogram files than with the HIST/FILE command.

```
zebra/rz/make  30 /home/nsdssn/costa/valerio.rz 1024 10000
* hist/file 30 /home/nsdssn/costa/valerio.rz 1024 N
nt/new 10 cosmic //lun30
....
*** fill ntuple here
....
cdir //lun30
hrout 10
....
....
close 30
```

As was mentioned above, variables can be specified with wild cards. This is most useful when defining ntuple variables with the NTUPLE/VARIABLE command. If no **tag** is specified the TAS variable name is used for the ntuple tag name (truncated to the 8 character limit in HBOOK).

Examples:

- **NTUPLE/VARIABLE tof.abar**
  Will add a variable with tag name ABAR to the **current** ntuple.
- **NT/VAR tof.a***
  Will add three variables with tag names ADC_T ADC_B and ABAR to the **current** ntuple.
- **NT/VAR tof.deltat tdif**
  Will add the tof.deltat variable to the **current** ntuple with a tag name of TDIF.
- **NT/VAR tof(front).a* f**
  Will add three variables to the **current** ntuple with tag names of FADC_T, FADC_B and FABAR. When a **tag** is specified along with a wild card variable specification, **tag** is prefixed to the variable name.
- **NT/VAR parts.* s**
  Will add all (5) variables from the parts table as 8 ntuple variables with tag names SIP, SP41, SP42, SP43, SP44, SIA, SIZ, and SMASS. Since the parts.p4 variable is a vector, each component becomes an ntuple variable and the index number of the component is post fixed to the tag name.

- **NT/VAR parts.p4(4) energy**
  Will add the fourth component of the parts.p4 variable to the **current** ntuple with the tag name ENERGY.

2.4. Table filters

Table filters are similar to plot filters and the same filter function routines can be used. Both types of filters use integer COMIS functions. There are some important differences and these will be discussed here.

A table filter can reference only one table so there is no joining of tables with various combinations of rows. The filter function is called once for each row of the table that contains valid data (rows 1 to NOK). If the row passes the filter (return value = 1) it is left intact. If the row fails the filter (return value = 0) the values are actually deleted from the table, the higher rows are shifted down and the table's NOK counter is set accordingly.

The other return values that are listed in the section on "User function status codes" are checked for table filters and they can cause actions to abort processing the current event (-2), abort the event processing loop (-4), and even force program termination (-6).

Table filters can be called at various times during the event processing. This is specified with a **when** expression as described in a following section.

2.4.1. commands

The single TAS command
DEFINE/FILTER_TABLES act when cfunc
is used to manipulate table filters. The particular action taken by the command is specified by **act**, the acceptable values are:

- **ADD** - add filter
- **DEL** - delete filter
- **ON** - switch filter on, if it was off
- **OFF** - switch filter off
- **MOVE** - move **when** the filter is called from one time to another
- **SHO** - lists the current state and definitions of all table filters.

An example is adding a filter to pass only alpha particles in the parts table after a call to the FREESCO event generator analysis module (FREESCO fills the parts table).
DEF/FILT add freesco ialphas.for(parts.ia,parts.iz)
The Fortran file ialphas.for is

```
       integer function ialphas(ia,iz)
c
c  A filter function to pass only alpha particles.
c
       ialphas = 0
       if ( iz .ne. 2 ) return
       if ( ia .ne. 4 ) return
       ialphas = 1
       return
       end
```

This filter function will be called after the analysis module for the FREESCO event generator and for each particle produced by FREESCO (each row of the parts table). If the particle is not an alpha particle it will be deleted from the parts table. The example parts table from the Introduction will now look like:

Parts Table after ialphas filter.

Variables / Values					
row	ip	p4	ia	iz	mass
1	4	(-.45,.51,3.67,5.275)	4	2	3.727

with it's NOK counter set to one.

2.4.2. miscellaneous comments on filters

The arguments to table filter functions are the actual TAS variables so that a table filter can modify a value in a TAS table. This permits interactive modification of the way variables are calculated which is convenient for calibrations, histogram weight variables, etc.

One is able to access global variables in these COMIS functions by using KUIP vectors.

The maximum number of arguments allowed for filter functions is 10, this is set by COMIS.

It does not work to pass character variables to filters.

2.4.3. when expressions

Table filters can be called at the start of processing an event (after getting a new event), after any analysis module and at the end of processing an event (before plot filling). Exactly when a table filter is called is specified by the **when** expression of the DEF/FILT command. The format of a **when** expression is **mod.n** where **mod** is FIRST, LAST, an analysis module name or an integer and **n** is an integer.

When **mod** is an integer it is taken to be the analysis module number. The analysis module number is listed by the DEF/FILT SHO command. **mod** =0 is the same as **mod** =FIRST.

The **.n** portion of a **when** expression is sometimes optional, as in the ADD example above. When **.n** is given it specifies which one of several filters that are specified for the same **mod**. Filters with lower values of **n** are called before higher values of **n**, for the same **mod**.

In detail

- **when** = FIRST means that the filter is called after TAS reads in a new event and after the call to TAE_USER at beginning of event. This call is made from the Processing Manager level in TAS which, in the future extension for distributed event processing, will be a separate process from the Event Processor, where other table filters are called.
- **when** = LAST means that the filter is called after all analysis modules but before the call to TAE_USER at the end of event, before plot filling and before writing output data. This is also called from the Processing Manager.
- **when** = sample means that the filter is called after the sample analysis module regardless of whether or not the analysis module itself was called. This filter is called from the Event Processor.
- **when** = 4.2 refers to the second filter called after the fourth analysis module.

2.5. User function status codes

There are various places where status codes are returned from user routines and influence program execution. These codes are checked under the following conditions and the appropriate action taken:

- the return value of TAE_USER
- the return value of each analysis module
- the return value of table filter functions

These codes are specified in include file `tas_user_codes.inc`. This file can be included into analysis modules. Since COMIS does not have include files, table filter functions must specify the actual numeric value of the status code. Because of this, these PARAMETER values are not expected to change.

```
!     TAS_USER_CODES.INC            6-6-90      D.O.
!
!  Contains parameters for status codes returned from user routines.
!
      INTEGER*4 tucod_ok
      INTEGER*4 tucod_fail
      INTEGER*4 tucod_abevt
      INTEGER*4 tucod_abgo
      INTEGER*4 tucod_exit
      INTEGER*4 tucod_unkmode
!
      PARAMETER (tucod_ok = 1)       ! general success status
      PARAMETER (tucod_fail = 0)     ! general failure status
      PARAMETER (tucod_abevt = -2)   ! abort processing event
      PARAMETER (tucod_abgo = -4)    ! abort GO event looping
      PARAMETER (tucod_exit = -6)    ! force program termination
      PARAMETER (tucod_unkmode = -123456) ! unknown mode in call to TAE_USER
```

2.6. Analysis modules

Analysis modules are called in the order they are declared in the module interface definition file. Exactly which analysis modules are called for processing a given event depends on a number of things.

TAS has a 3 position switch for each analysis module with the states of ON, OFF and AUTO.

These are set by the command
SET/MODULES name [state]
where **name** is the name of an analysis module as declared in the .mid file (wild cards allowed) and **state** is ON, OFF or AUTO.

- **state** = ON means this analysis module is always called
- **state** = OFF means this analysis module is never called
- **state** = AUTO (the default state) means this module is called if it is needed to produce the requested output.

TAS checks which tables are filled, after getting a new event, and compares these to the list of tables which are requested for output, i.e., for plots, output data files (save table list), or the manual table fill list. If any of the requested tables are not filled, the analysis module - table dependencies, are checked to determine which modules to call. These modules (except those switched OFF), along with any that are switched ON, are called in the order of declaration in the .mid file.

2.7. Data I/O

TAS provides for two kinds of data files, event stream files and table files.

Event stream data is the normal event data that gets processed from raw data into summary physics data. An event stream file can consist of raw data or processed data consisting of tables. One is able to save the data at any stage of processing in the output event stream and read it in later for further processing. The commands used for event stream data files are:

- **DATA_IO/INPUT_DATA source [type lrec]**
  Opens an input event stream, data file or live data.

- **SET/SAVE_TABLES action [tablist]**
  Used to generate the list of tables to save in the output event stream file. This command must occur before opening an output event stream file.
- **DATA_IO/OUTPUT_DATA [name type lrec]**
  Opens an output event stream data file.
- **DATA_IO/CLOSE inout**
  Closes either input or output event streams.

Table data files are used for calibration constants, switch settings and the like and do not have an event structure. The commands used for table data file are:

- **DATA_IO/READ file [type]**
  Opens, reads and closes a table data file.
- **DATA_IO/WRITE file type tables**
  Opens, writes and closes a table data file.

In addition to these two kinds of data files there are different formats specified by **type**:

- **type** = FZN means Zebra native binary format. Note that this format is NOT machine independent
- **type** = FXZ means Zebra binary exchange format.
- **type** = FZT means Zebra binary exchange tape format (using the cf routines).

2.8. Calibration file procedures

This section describes the procedures included in TAS to facilitate handling calibration constants which vary from run to run.

Notation used for the following discussion is `ALL_UPPER_CASE` refers to logical names on a VAX and environment variables on UNIX and `table.var` refers to a TAS table variable.

The variables used as part of the calibration management procedures are

- `evnt.run` the current run number
- `calfiles.file` the name of a calibration file
- `calfiles.runlo` the lower limit of run number for this file
- `calfiles.runhi` the upper limit of run number for this file
- `calfiles.table` the table name to check for this file
- `calfiles.type` the file format type (FZX or FZN)
- `calfiles.mode` the read mode for the file, replace or append
- `table_desc.h.mod_count` this is not a TAS table variable as such but is a flag for each table (like `table_desc.h.nok`) that gets incremented each time a static table is modified and is set to zero at a begin run.

A command TAS/CALIB/ADD is used to add entries to the `calfiles` table. This command is described in the section for the on-line manual.

The calibration files listed in the `calfiles` table are of two types, those that apply to a specific table and those that always get read in at a begin run. If the `calfiles.table` variable is blank then the associated file is always read in at begin run. If the `calfiles.table` variable is not blank then the file is read in if both of the following conditions apply (this check is performed each event):

- `table_desc.h.mod_count` is zero for the table named `calfiles.table`
- an analysis module will be called that has declared read or update access to this table

If calfiles.file is blank but `calfiles.table` is not blank then the above procedure still applies with the file name constructed as
`TAS_CALIBS:run####_calib.type`
where `TAS_CALIBS` specifies a directory for calibration files and `####` is filled in with the run number from `evnt.run` and `type` is taken from `calfiles.type` (or `fzx` if `calfiles.type` is blank).

The various actions performed depending upon the entries in the `calfiles` table are listed in the table below. These actions are taken for each entry (row) in the calfiles table.

calfiles variables[a]						action
runlo	runhi	file	table	type	mode	
*	*	*	-	*	R	file is read at the beginning of a run with runlo ≤ run ≤ runhi and replaces previous contents of the tables
*	*	*	-	*	A	file is read at the beginning of a run with runlo ≤ run ≤ runhi and appends to the previous contents of the tables
*	*	*	*	*	*	file is read on the first event after the beginning of a run with runlo ≤ run ≤ runhi in which an analysis module will be called that has read or update access to table and table has not yet been filled (table_desc.h.mod_count=0)
0	0	*	-	*	*	file is read at the beginning of any run
85	85	-	-	-	-	TAS_CALIBS:run85_calib.fzx is read at the beginning of run 85 and replaces the previous contents of the tables
23	56	-	-	-	-	TAS_CALIBS:run42_calib.fzx is read at the beginning of run 42 and replaces the previous contents of the tables

[a] Notation:
* means any non-blank or non-zero value
- means blank

3. Programming Guide

Most of the programming practices required to use TAS are enforced automatically by the manner in which analysis routines are attached to the shell and by data structures management system (see Appendix A).

All specific analysis routines (analysis modules) are connected to TAS by using the `tas_gen` procedure which reads an interface definition file and generates the code which specifies what tables and what analysis modules will be included in a particular executable program.

An additional user entry point provided for situations that do not correspond to an analysis module is the `tae_user` routine. This is described in more detail below.

At this point it is necessary to explain some technical details about how tables are handled in the program development procedure. Since tables are objects[8] which need to be initialized in order for the shell to understand them, the initialization routines must be managed in some fashion. The initialization procedure used with TAS requires that sets[9] of tables be identified with particular libraries of source and object code. Code from any library can reference a table but a given table must be "owned" by only one of the libraries used in building an executable program. This "ownership" is specified by the connection between a library and a table in the table editor (see Appendix A).

3.1. tas_gen interface generator

The `tas_gen` procedure is used to generate the source code which connects analysis modules and tables to a given executable program[10]. The input file to this procedure is called a module interface definition (.mid) file. An example of this for the txm_amsamp module is:

```
# definition file for txm sample TAS program
#
table_lib txm
#
module       sample txm_amsamp
#
module_table sample samp_sw     update  1
module_table sample mytable     write   2
```

In the format of a `.mid` file a comment line begins with a # character and all other lines begin with a keyword followed by parameters. The keywords and parameters are:

- **table_lib** `lib`
  `lib` is the name of a library which "owns" a set of tables
- **module** `name function`
  `name` is the character string which is used to identify a particular analysis module
  `function` is the actual name of the function which is the top level routine for this analysis module
- **module_table** `module table access_mode position`
  `module` is the name of the analysis module this table is associated with[11] (=**module** `name`)
  `table` is the name of the table

---

[8] The term object is used loosely here since TAS tables to not have all the properties required to qualify them as objects in an object-oriented language like C++ or Smalltalk.

[9] A set of tables can be one or more tables.

[10] This is very similar to the CDF Analysis Control BUILD_JOB procedure.

[11] A table can be associated with an arbitrary number of analysis modules.

`access_mode` specifies whether this module will have read, write or update[12] access to this table
`position` is the relative position of this table in the argument list in the function call to this module

The order of lines in the .mid file is arbitrary except for the **module** keywords. The order of modules in the .mid file determines the execution ordering of these modules during event processing. This aspect of the .mid file will probably be changed in a future version so that the execution order of modules will be adjustable at run time. Also note that the `module` parameter of the **module_table** keyword is matched with the name `parameter` of the **module** keyword.

For a module interface definition file named *`defn`*`.mid`, the `tas_gen` procedure generates an output file called `tas_`*`defn`*`.f`. This Fortran source file contains the set of routines which connect tables and modules to the shell and it must be compiled and linked along with the TAS library and your analysis module code to make an executable program.

The format of the module interface definition file is expected to evolve in future versions by having additional keywords and include files.

3.2. Style guidelines

There are a few programming considerations which help analysis routines fit in with both the present and future aspects of TAS. These are:

- include files should be specified as
  `INCLUDE 'filename_inc'`
  where the file specification should be in lower case. The `filename_inc` name should be equivalenced to a file named `filename.inc` by the use of symbolic links on UNIX and logical names on VMS.
- individual analysis routines (analysis modules in TAS parlance) should not do any direct I/O for either data or control. The static tables in TAS are to be used for both calibration constants and miscellaneous control features required of these routines.

3.3. Analysis modules

The philosophy of an analysis module is that it is a routine (or collection of routines) that performs operations on tables. It reads one or more tables and writes or updates other tables.[13] I/O and plot filling are handled by the TAS shell to and from the tables.

The code that interfaces a particular analysis module to the shell is generated by the `tas_gen` procedure according to the specification in the module interface definition file (see `tas_gen` above). Each analysis module is executed as an integer function with table headers and table rows as pairs of arguments (see the `txm_amsamp` example below). The function value returned is interpreted by the shell as a user function status code (described above).

The example below shows the standard interface for an analysis module to access tables. This standard interface is designed to permit the evolution of tables to be allocated from dynamic memory rather than statically allocated in a Fortran common block. A non-standard (but functional) method of accessing tables which are defined in common blocks is to reference the common blocks directly in the analysis module code via the `lib_table_com.inc` include file which is generated by the data structure management system.

3.3.2. TXM_AMSAMP

This sample analysis module corresponds to the module interface definition file described above in the section on `tas_gen`. It is important to note the relationship between the actual argument list and the values

---

[12] Update implies both read and write access.

[13] Reading or writing tables is strictly philosophical since the tables are passed as arguments in the subroutine call to an analysis module.

of the **module_table** `position` parameter in the .mid file. As mentioned above, each table specified in the .mid file yields two arguments to the module. The first of these is a `table_head_st` record that contains some book keeping parameters for the table and the second argument is the row data for the table. The structure name for the *`table`* row argument is *`table`*`_row_st` as seen below. The table rows argument contains the data and all the variables that you defined in the data structure management system.

There are three important parameters in the `table_head_st` argument each of which is illustrated in the example below. These parameters are (where `head` refers to a generic `table_head_st` record):

- `head`**`.maxlen`** = the maximum number of rows allocated for the table. This parameter is set by the code generation procedures and should **NEVER** be modified by the analysis module.
- `head`**`.nok`** = the number of rows in the table containing valid data. This should be set to the appropriate value by the module and should be checked to see that it never exceeds head.maxlen.
- `head`**`.mod_count`** = a flag which gets incremented whenever a static table is modified. The shell increments this flag for I/O and interactive operations which modify a static table and it should be incremented by the analysis module if the analysis module modifies a value in a static table. The analysis module can also check the value of this parameter to see if values in a static table have been modified since the last call to the module in order to update any local values which depend upon the static table.

```
      INTEGER FUNCTION TXM_AMSAMP( swh, sw, myh, mytable )
      IMPLICIT NONE
C   Input arguments  (One to a line with definition after ! )
C     swh = header for samp_sw table
C     sw = rows of samp_sw table
C     myh = header for mytable
C     mytable = rows of mytable table
C
C   Output arguments :
C     values in sw, mytable
C     iok = user status code
C
C   Functional Description :
C     a sample analysis module
C   Created  7-Jan-1993   Doug Olson
C   Copyright 1993    Lawrence Berkeley Laboratory
C   Error conditions :
C-----------------------------------------------------------------------
      INCLUDE 'tas_structures_inc'
      INCLUDE 'tas_user_codes_inc'
      INCLUDE 'txm_samp_sw_pars_inc'
      INCLUDE 'txm_samp_sw_st_inc'
      INCLUDE 'txm_mytable_pars_inc'
      INCLUDE 'txm_mytable_st_inc'
C-----------------------------------------------------------------------
      RECORD/ table_head_st/ swh
      RECORD/ samp_sw_row_st/ sw
      RECORD/ table_head_st/ myh
      RECORD/ mytable_row_st/ mytable(*)
      INTEGER i, loop, nok
C-----------------------------------------------------------------------
C  check switch A, if set to 1 then zero mytable data (a nonsense thing to do)
      if(sw.a .eq. 1) then
         do i = 1, myh.maxlen
            mytable(i).c = ' '
            mytable(i).i = 0
            mytable(i).r = 0
         end do
C
C  set counter for number of rows of good data to 0
         myh.nok = 0
C
C  set return code to normal success
         txm_amsamp = tucod_ok
C
C  set switch A back to 0 and increment the modified counter for since it
C     is a static table
         sw.a = 0
         swh.mod_count = swh.mod_count + 1
         goto 999
      end if
```

```
C
C do something else useless *********************
C make sure the loop counter does not exceed the maximum number of rows
C allocated for mytable
      if(sw.c .gt. 0) then
         loop = min( sw.c, myh.maxlen)
      else
         loop = myh.maxlen
      end if
C
C set the counter for the number of rows of good data to zero
      nok = 0
C
C loop over the rows of mytable
      do i = 1, loop
C
C make sure the row counter (nok) does not exceed the allocated space
         if(nok .lt. myh.maxlen) then
C
C increment the row counter
            nok = nok + 1
C
C make some kind of calculations
            if(mod(mytable(nok).i,sw.b) .eq. 0) then
               mytable(nok).i = 0
            end if
            mytable(nok).i = mytable(nok).i + 1
            mytable(nok).c = 'hello'
            mytable(nok).r = 3.14 * mytable(nok).i
         end if
      end do
C
C set the row counter in the descriptor structure for mytable
      myh.nok = nok
C
C return a normal success code
      txm_amsamp = tucod_ok
  999 CONTINUE
      RETURN
      END
```

3.4. TAE_USER(mode,tptrs,ntables)

This routine (an integer function) is called by TAS at various times, but always from the processing manager level rather than the event processor level (see the functional description of TAS below). This routine is provided for additional code that does not fit into the analysis module framework. An array with all tables, is passed as the second argument, `tptrs`. The number of tables in this array is given by the `ntables` argument. The condition under which it is called is provided by the `mode` argument. These are:

- `tu_mode_init` - called once during program startup.
- `tu_mode_gobeg` - called after giving the TAS `GO` command but before the event processing loop.
- `tu_mode_evbeg` - called for each event during the event processing loop, after the TAS get event, before table filters with **when** = FIRST and before any analysis modules.
- `tu_mode_evend` - called for each event during the event processing loop, after all analysis modules, after table filters with **when** = LAST but before filling plots or writing event stream output data.
- `tu_mode_goend` - called after the event processing loop but before returning to the user interface level.
- `tu_mode_exit` - called once at normal program termination.

The philosophy of TAE_USER is that it handles all of the cases needed by analyzing data in the real world that are not handled by either the TAS shell or the analysis modules. For example, this is the appropriate

place to put alternate forms of input and output that TAS does not support.[14] One could also define and fill standard sets of histograms here and still be consistent with the distributed event processing extension to TAS.

An example tae_user routine is included with the TAS source and the TXM example analysis module source.

3.5. Use of tables

There are two types of TAS tables, static and dynamic, and they have some similarities and some differences. The similarities are:

- they have the same format for Fortran structures and data files.
- they can both be used to fill plots.

The differences are:

- filled flag
  - Dynamic tables have their filled flag reset to FALSE (0) at the beginning of each event (before reading a new event).
  - Static tables have their filled flag set to TRUE (1) whenever they are initialized or get values loaded by any of several means. The filled flags are not reset during the event processing loop.
- NOK counter
  - Dynamic tables have their NOK counters set to 0 before reading a new event (and they are set appropriately for the tables that are filled by reading an event).
  - Static tables are not reset each event, NOK is set only by loading tables (reading files, setting values manually, or copying an ntuple to a table).
- modified flag
  - Static tables have their modified flag (table_desc.h.mod_count) incremented automatically by TAS whenever their values are changed, either by reading a file or from the command line. One of the requirements of analysis modules is that if they change a value in a static table they must increment this modified flag also.
- saving tables in output data event stream
  - Dynamic tables are written out each event
  - Static tables are written just once for the first event

Values can be set in tables interactively with the command SET/VALUE. This is useful with static tables for calibration constants or switch settings. It is of limited value for dynamic tables since the NOK counter gets reset before calling any analysis modules, which effectively removes any values set interactively.

---

[14] In the case of the EOS executable program this is used for reading the SCA chip test data and writing the stick pedestal data files.

4. Functional description of TAS

TAS is primarily an event loop processing addition to PAW (see Fig. 1). TAS is connected to PAW via the KUIP user interface so that the TAS commands appear in addition to all of the PAW commands. The level called the Processing Manager handles plot definitions and filling, data I/O, and calling the event processor during the event loop. The Event Processor level calls the individual analysis modules and table filters. The necessary code that depends upon the definition of TAS tables is generated automatically as reports from the INFORMIX database holding the table definitions.

At the interactive level there are four stages to processing data. These are

- Initialization
  - specify data input
  - specify what output is to be generated
  - set any manual switches wanted for controlling analysis modules
  - load calibration constants
  - set individual values for analysis routines
- Open output data file
  - this is only if you have specified output tables to save
- Process events
  - handled by the processing manager level in TAS
- Finish
  - close input and output
  - look at plots and do interactive analysis with PAW

4.1. Architecture and future extensions

The architecture of TAS is designed so that the user interface (UI), processing manager (PM) and event processor (EP) operate by sending commands. Version 2.X of TAS operates as a single process but this design enables the future extension to having these three parts operating as separate processes. This enables distributed event processing and manipulating graphics displays asynchronously with the event processing loop.

Figure 2 shows a diagram of TAS operating in this fashion. The UI (with PAW) is the parent process. The PM is run as a subprocess and communicates with the UI process. The PM also handles the data I/O and distributing events to the EP processes. The EP processes may be on remote machines so that with a single control interface and a single I/O channel, events may be processed on a farm of workstations.

Figure 1.

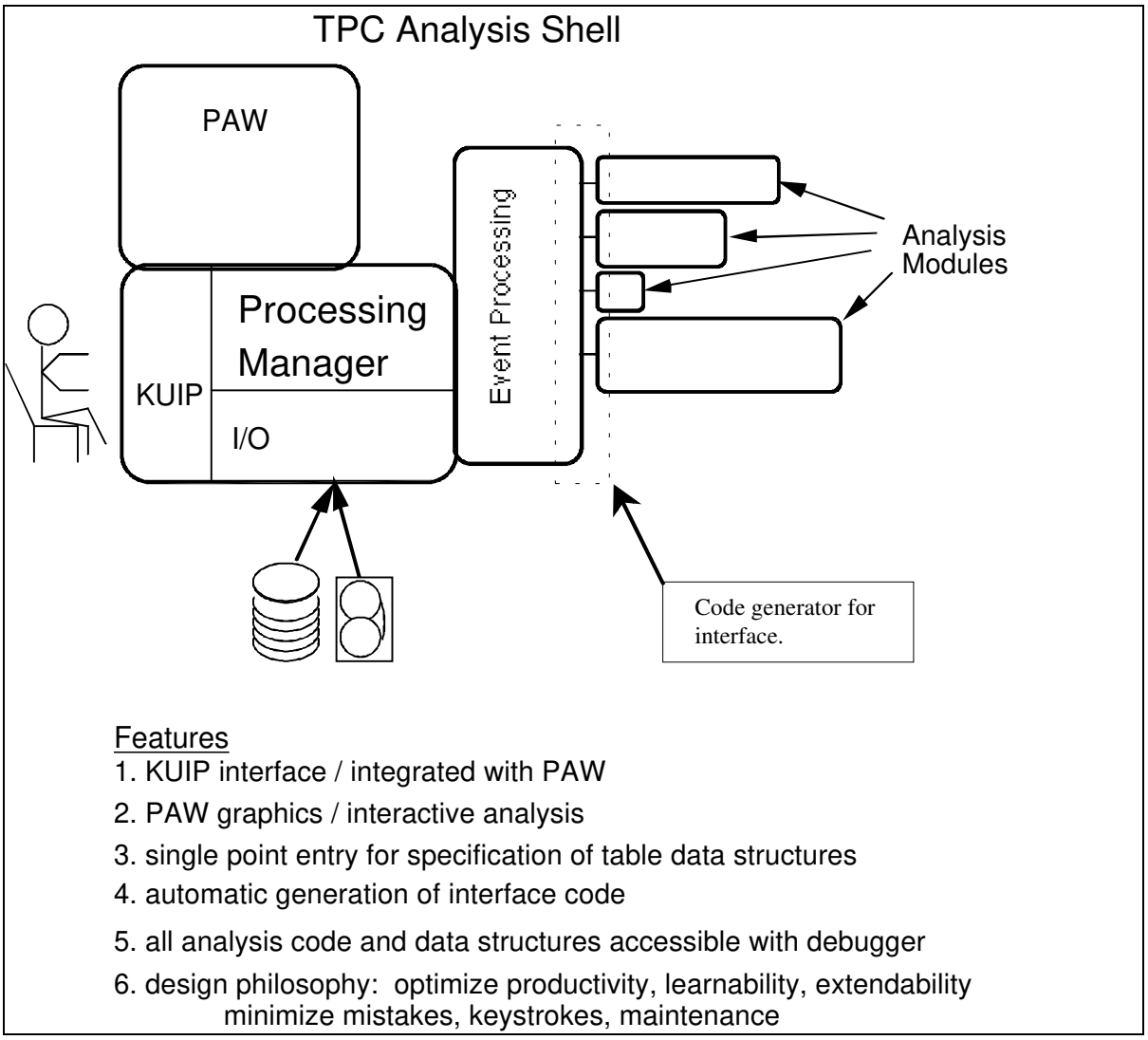
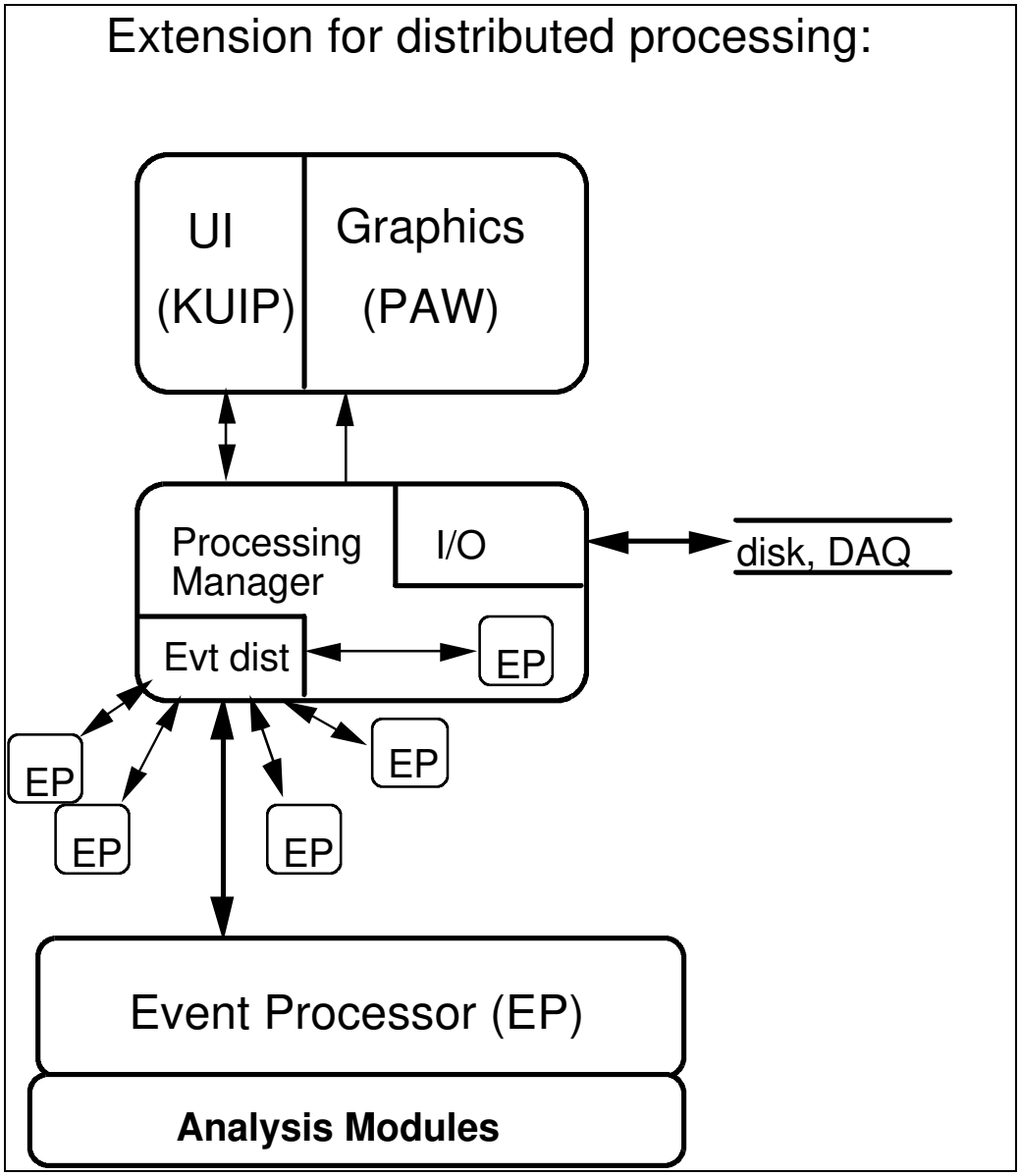

TPC Analysis Shell
PAW
Processing
Manager
KUIP
I/O
Event Processing
Analysis
Modules
Code generator for
interface.
Features
1. KUIP interface / integrated with PAW
2. PAW graphics / interactive analysis
3. single point entry for specification of table data structures
4. automatic generation of interface code
5. all analysis code and data structures accessible with debugger
6. design philosophy: optimize productivity, learnability, extendability
minimize mistakes, keystrokes, maintenance

Figure 2.


Extension for distributed processing:
UI
(KUIP)
Graphics
(PAW)
Processing
Manager
I/O
disk, DAQ
Evt dist
EP
EP
EP
EP
EP
Event Processor (EP)
Analysis Modules

4.2. Event processing, or what happens when you type GO

This section describes the sequence of actions that take place during the event processing loop. This is activated by the command GO. The description of these actions are divided up into the three levels UI, PM and EP.

The user interface (UI) sequence is:

1. KUIP calls the Fortran routine which handles GO.
2. send GO command to PM
3. *event processing happens here*
4. check return status for errors and report on the number of events processed.
5. return to KUIP

At the processing manager (PM) level in TAS the sequence of actions are:

1. call `TAE_USER(tu_mode_gobeg,,)`
   - continue, return or exit based upon return status
2. check output request and manual analysis module switch settings
   - return if there is nothing to do
3. check / set event loop counters
4. get event
   - set all filled flags to FALSE and all NOK counters to 0 for dynamic tables (static tables are ignored)
   - read in next event
   - break out of event loop on error
5. call `TAE_USER(tu_mode_evbeg,,)`
   - check return status and take appropriate action
6. call table filters with **when** = FIRST
   - take appropriate action based upon return status
7. call Event Processor
   Analysis modules and table filters are called here.
8. call table filters with **when** = LAST
   - take appropriate action based upon return status
9. call `TAE_USER(tu_mode_evend,,)`
   - take appropriate action based upon return status
10. fill plots
11. write output file
12. continue event loop (3)
13. call `TAE_USER(tu_mode_goend,,)`
    - take appropriate action based upon return status
14. return to UI

At the event processor level the actions are:

1. compare list of filled tables with those needed for output,
   check for table filters or analysis modules that are switched ON
   - return if nothing to do

2. call individual analysis modules
   - call analysis module
   - return if error code set
   - call table filters for this module (**when**)
   - return if error code set

3. return to PM level.

Appendices

A. Data Structure Management System (DSM)

Before discussing the details of the interaction with the data structure management system, DSM, (currently implemented with an INFORMIX database) it is worthwhile to give an overview of what a TAS program is and how it related to the definitions of TAS tables. A TAS application program is defined by a the type of data the program can access (TAS tables) and the code that operates on that data (analysis modules). The set of data structures (TAS tables) that are accessible in the program are managed with the data structure management system. A single TAS application program corresponds to a specific set of TAS tables, each table having a certain dimension of the maximum number of rows allocated in memory. Two different TAS application libraries can have TAS tables with the same name (which means identical definitions for the variables) but with different dimensions for the maximum number of rows. Also, different TAS applications can have different TAS tables, they need not have the identical set of tables.

The set of tables used in an application program is determined at compile time by the source files generated by the `tas_gen` procedure (see above) which selects which application libraries will be included. The application libraries have the include files generated by the DSM. These files are included in a number of other source files, all of which get compiled and loaded into an application object library, specifically for that program.

The database referred to in this document as INFORMIX consists of a collection of products by the INFORMIX company and various forms and reports developed by D. Olson to handle the various aspects of defining TAS tables and generating the appropriate source code.

A few terms need to be defined for this discussion:

- form
  A screen oriented display with various fields and labels that provide access to information stored in the INFORMIX database. The table editor discussed in a following section is a form.
- report
  A formatted listing of information selected from the INFORMIX database. The Fortran include files that define the TAS table data structures are INFORMIX reports.
- INFORMIX table
  The information stored in the INFORMIX relational database is stored as tables which are similar to the TAS tables (actually TAS tables are similar to INFORMIX tables). They consist of a set of rows, each row has several variables. A clear distinction needs to be made between INFORMIX tables and TAS tables when using the Table editor.
- TAS table
  The data structures used in TAS and whose definitions are stored in the INFORMIX database.
- TAS application library
  A library built with a specific set of TAS tables, each with a specified maximum dimension for the number of rows.
- Current list
  The list of INFORMIX table rows selected by the previous Query command.
- active table
  The INFORMIX table in a form which is accessible for modifications.

Defining a new TAS table or modifying an existing one is a two step process. First one uses the Table editor to make the definitions and then one executes the procedure that runs the reports to generate new source code.

A.1 Table editor

The TAS table editor is started with the command `tas_edit`. This brings up a form which accesses the INFORMIX tables containing the definitions of the TAS tables. The initial screen display looks like:

```
PERFORM:   Query  Next  Previous  View  Add  Update  Remove  Table  ...
Searches the active database table.              1: tas_libs table
Hints:  Ctrl-D = delete field,  Ctrl-X = delete char, Ctrl-R = refresh screen
Ctrl-[ = Escape, Ctrl-C = Interrupt,  Ctrl-Z = Don't do it!,  Delete = nothing
----------------------  TAS Libraries  ----------------------------------------
Library name: [                    ]  Owner: [                     ]
 description: [                                                          ]
              [                                                          ]
----------------------  Tables  -----------------------------------------------
        Table:                          Max length:
   Zebra code:
   Descripton:
      more...:
   Table Type:
   Created by:                          at:
  Modified by:                          at:
----------------------  Variables  --------------------------------------------
         Name:                           Data Type:
      Comment:
Vector length:                Initial value:
Created Btime:                Modified Btime:
--------------------------------------------------------------------------------
```

The first line of the screen is standard to all INFORMIX forms and is the list of available INFORMIX commands. The second line contains a brief description of the currently highlighted command and indicates which INFORMIX table is "active", (tas_libs above). The "active" INFORMIX table is also indicated by square brackets, [], around the corresponding fields in the form. The next two lines describe some editing keystrokes of use in operating the form.

The next section titled ``TAS Libraries" is for the INFORMIX table (tas_libs) which has the list of application libraries that have sets of tables from this database.

The section called ``Tables" is used for both the list of TAS tables belonging to a particular library (libtabs INFORMIX table) and for the header information of TAS tables which is stored in the INFORMIX table tast_h. The bottom section titled ``Variables" is for information about the variables in TAS tables which is stored in the INFORMIX table tast_v.

The INFORMIX commands for performing actions are listed in the top row. The action is performed by typing the first letter, Q for Query, N for Next, etc., or selecting with the cursor keys and hitting `<return>`. A comment on what the command does is listed in the second row (``Searches the active database table." in the example above). The commands and comments are:

- **Query** - Searches the active database table.
- **Next** - Shows the next row in the Current List.
- **Previous** - Shows the previous row in the Current List.
- **Add** - Adds a row to the active database table.
- **Update** - Changes a row in the active database table.
- **Remove** - Deletes a row from the active database table.
- **Table** - Selects the current table.
- **Screen** - Shows the next page of the form.
  (unused by `tas_edit`)
- **Current** - Displays the current row of the current table.

- **Master** - Selects the master table of the current table.
  (unused by `tas_edit`)
- **Detail** - Selects a detail table of the current table.
  (unused by `tas_edit`)
- **Output** - Outputs selected rows in form or report format.
  (not very useful with `tas_edit`)
- **Exit** - Exits the table editor.

The next two lines describe some editing control keys. `Ctrl-D` deletes within a field from below the cursor and to the right. `Ctrl-[` is the escape key, useful on VAX keyboards that do not have a dedicated escape key. `Ctrl-C` will cancel any action. `Ctrl-Z` will suspend the process and put it in the background. You should be careful not to do this since INFORMIX does not recover well from this. The delete key does nothing, you should use back-space or left-arrow to delete characters to the left. If the screen gets garbled a `Ctrl-R` will refresh it.

The fields in the TAS Libraries section are:

- **Library name**: The name of the object/source code library which will have a particular set of tables associated with it. This library name is used as a prefix in filenames generated from the database.
- **Owner**: the name of the user who first added a new record for a library.
- **Description**: a brief description of this library.

The fields in the middle section, for TAS table header information, are:

- **Table**: The name of the TAS table. It should be a valid Fortran name (no spaces or funny characters). It is used as a root name for the structure and parameter names associated with this table.
- **Max length**: The maximum number of rows allocated for this table. It is the dimension of the row array for the table. This field applies to a particular TAS application program and can be set differently for different applications (for the same table name).
- **Zebra code**: A unique 4 letter code used for the Zebra IDH bank descriptor in the Zebra I/O subsystem. This code is not used in any user level part of the shell.
- **Description**: and **more...**: A brief description of the TAS table. This is put into the `tas_table_st.inc` file as a comment.
- **Table Type**: S = static table, D = dynamic table
- the following fields are filled automatically
  Created by: The username of the person who Added this table.
  Created on: The integer time stamp of when this table was Added.
  Modified by: The username of the last person who Updated this table (this row of the `tast_h` table).
  Modified on: The integer time stamp for when this row of the `tast_h` table was Updated.

The ``Variables'' section of the screen shows the fields which define the individual variables in TAS tables. These fields are:

- **Name**: The name of the current variable in this table. This must be a valid Fortran name.
- **Data Type**: The data type of the variable, I for integer*4 or R for real*4, C for Character, D for Real*8
- **Comment**: A description of this variable. This is written into the `tas_table_st.inc` file as a comment.

- **Vector length**: The dimension of a vector variable. The default value is 0 corresponding to a scalar variable.
- **Initial value**: The initial value for this variable in every row of the TAS table. This applies only to static tables. The default value is 0.
- the following fields are set automatically
  Created Btime: The integer time stamp for when this variable was Added.
  Modified Btime: The integer time stamp for when this variable was Updated.

A.1.1 editing fields

The fields (variables) in the active table are surrounded by square brackets ([]). The active INFORMIX table is changed by typing T (the INFORMIX Table command). This is used to cycle through the different sections of the form.

The editing style of INFORMIX forms is different from a standard VMS style. The actions of editing keys are described below.

The keystroke terminology for the remainder of this discussion is as follows:

- an alphanumeric keystroke is denoted as the character itself, i.e., `A` for the letter A key.
- non-alphanumeric keys are denoted by `<keyname>`, i.e., the control key is `<ctrl>`.
- a sequence of holding down one key while typing another is denoted by a dash (-), i.e., control-A is denoted `<ctrl>-A`.

The following keys perform special actions within a form:

- `<TAB>`, `<shift>-<TAB>`, `<UP>`, `<DOWN>` and `<return>` move the cursor from one field to another
- `<ESC>` completes an editing action in a form and makes the changes to the database
- `<ctrl>-C` cancels the action without making changes to the database.
- `<ctrl>-Z` on UNIX has the very bad effect of pushing the INFORMIX process into the background and there does not seem to be a way to recover from that (fg does not work). DO NOT TYPE `<ctrl>-Z`.

The following keys perform editing actions within a field:

- `<ctrl>-A` toggles between overtype and insert modes
- `<ctrl>-X` deletes the character under the cursor
- `<ctrl>-D` deletes from the character under the cursor to the end of the line
- `<backspace>` and `<LEFT>` both move the cursor one character to the left without deleting anything
- `<delete>` has no effect (a flash and a beep but no editing function)
- `<RIGHT>` moves the cursor one character to the right without deleting

A.1.2 example

The following sequence will define a new dynamic table called `mytable` with 4 variables, `i`, `r`, `c`, `vec`, and 10 rows in the `txm` application library. Things you type are denoted in **bold face**. Starting from the DCL command line:

- **tas_edit**
  Starts the INFORMIX TAS table editor form.

- **Q**
  Enter Query mode for the `tas_libs` INFORMIX table.
- **txm <ESC>**
  Enter the name of the application program and do the query.
- **TT**
  Type T twice to select the `tast_h` table.
- **A**
  Begins the Add row action for the `tast_h` table.
- **mytable <TAB>**
  Enters the table name into the Root Name field and moves the cursor to the Max Length field.
- **MYTB <TAB>**
  Enter the Zebra code. This is used for the IDH of the Zebra banks for the FZ file formats.
- **a sample table <TAB> <TAB>**
  Enters a comment into Description and advances the cursor to the Table Type field.
- **<ESC>**
  Leaves the Table Type as the default (Dynamic) and enters this information into the database.
- **T**
  Switch to the `tast_v` table to enter the variable definitions.
- **A <TAB>**
  Begins the Add row action for the `tast_v` table and moves the cursor to the Name field without changing the Root Name (mytable).
- **i <TAB>**
  Enters the variable Name (i) and moves to the Data Type field.
- **i <return>**
  Enters the data type as I (integer*4) and moves to the Comment field.
- **an integer <DOWN>**
  Enters a comment and moves to the Vector Length field.
- **<ESC>**
  Completes the definition of ivar without setting a vector length or an initial value.
- **A <TAB> r <TAB> r <TAB> a real number <ESC>**
  Enter the definition for REAL*4 r ! a real number.
- **A <TAB> c <TAB> c <TAB> a character <TAB> 5 <ESC>**
  Enter the definition for CHARACTER*20 c ! a character.
  Note that the character size is 4*Vector length.
- **A <TAB> vec <TAB> r <TAB> vector <TAB> 4 <ESC>**
  Enter the definition for REAL*4 vec(4) ! vector.
- **TT**
  Type T twice to select the `libtabs` INFORMIX table.
- **A**
  Enter add mode to add this table to the `txm` application library.
- **txm <TAB> <TAB>**
  Enter the application name and move to the `Max length` field.

- **10 <ESC>**
  Enters 10 into the Max Length field and adds this record.
- **E**
  Exit the table editor.

A.2 Code generation

Once you have entered your table definitions you can look at the report files by executing the command **all_tas_reps.** This will generate several files including txm_mytable_st.inc which contains the data structure definitions for the TAS tables.

```
!    txm_mytable_st.inc
!     (tast_inc.ace)
!
!-------------------------------------------------------------------------
!   Table: mytable
! table descriptor name: mytable_desc_st
!    row structure name: mytable_row_st
!           description: a sample table
!
       STRUCTURE /mytable_desc_st/
           RECORD /table_head_st/ h
           RECORD /table_vinfo_st/ vinfo(mytable_nv)
       END STRUCTURE ! mytable_desc_st

       STRUCTURE /mytable_row_st/
           CHARACTER*20 c                  ! a character
           INTEGER*4 i                     ! an integer
           REAL*4  r                       ! a real number
           REAL*4  vec(4)                  ! vector
       END STRUCTURE      ! mytable_row_st
!  Last mod. for mytable:   729249171 Tue Feb  9 01:12:51 1993
!
```

Index

.mid file, 16
1-D histogram, 7
2-D histogram, 8
2D/FILT, 9
2D/VAR, 9
ADAMO, 3
analysis module, 3
application program, 26
C structs, 4
Calibration file procedures, 14
COMIS functions, 11
command line format, 6
Contents, 1
data format, 3
data processing, 3
data types, 7
dynamic tables, 20
event stream, 3
filter function, 11
Filters, 5
help, 6
Introduction, 3
KUIP, 6
kumac, 6
last.kumac, 6
macro, 6
maximum number of arguments, 12
mode, 4
mode of access, 4
module interface definition, 16
NOK, 4
ntuple, 8
ntuples, 9
on-line manual, 6
pawlogon.kumac, 6
plot filter, 5
Plots, 5
Programming Guide, 16
read, 4
return values, 9
row selections, 8
static tables, 20
STRUCTURES, 4
Table editor, 27
Table filter, 5
Table filters, 11
tables, 3
TAS data structures, 3
tas_gen, 16
taslogon.kumac, 6
TXM_AMSAMP, 18
update, 4
usage, 6
User function status codes, 12
User interface, 6
when expressions, 12
write, 4