GRANITE Data Format, *Reference Manual*

Version 0.76, August 1997

Requests for information should be send to:

```
Joachim Rose
Department of Physics and Astronomy
Leeds University
Leeds, LS2 9JT
United Kingdom
e-mail:  h.j.rose@leeds.ac.uk
```

Acknowledgements

The data format used by the Granite Collaboration has evolved over many years. Naturaly quite a number of features of the present version are derived from the previous versions. The Fortran90 derived types are similar to the VMS Fortran structures in the Granite data management software by Michael Schubnell [**?**, **?**].

Most of the data to be stored is produced by the CAMAC data acquisition software written by Glenn Sembroski. Other sources of data are for example the high voltage systems, the tracking computers or the CCD cameras. Many collaborators were involved in defining approiate derived data types for each of these data sources.

The structures for Monte Carlo data follow the conventions inside the MOCCA program written by Michael Hillas.

Although not visible from to the calling program GDF relies on the package ZEBRA [**?**]. Memory allocation is handled by Zebra-MZ [**?**] while data is stored in tape or disk files by calling Zebra-FZ [**?**] routines.

How to get GDF

The GDF source code, the object code libraries for different operating systems, this manual, and more information on GDF are available on

`http://phyvs1.leeds.ac.uk/whipple/gdf`

Please report any errors and problems to Joachim Rose, `h.j.rose@leeds.ac.uk`. Comments and suggestions on how to improve the software or this manual are also very welcome.

About this manual

This document has been produced using LaTeX [**?**] together with the `CERNMAN` style option, developed at CERN by Michell Goossens.

Table of Contents

List of Figures

Chapter 1: A tutorial introduction

This chapter serves as a quick introduction. The aim is to show the essential elements of the package in real programs, but without getting lost in details, rules and exceptions. The example programs in this chapter are not trying to show all the features and possibilities. All examples do work and can be used as a starting point to build useful programs as quickly as possible. Therefore this section concentrates on the basics: initialisation, opening of files for reading, and access to the data via derived type data structures in memory.

The main program below calls different subroutines depending on how many parameters were given on the command line.

- If no parameter is given the routine GDF$MANAUL produces the GDF manual as a LaTeX file. The master file is `GDF.TEX` which contains references to other files that are also generated.
- If a single file name is given when invoking the program the several examples subroutine are called which each read the input file.
- Two files names on the input lines will cause two calls to GDF$TEST. The first call to write a new test data file, the second call one to read a test data file and check that its contents are read correctly. The second file can be different from the first and may originate from a different type of computer. This makes it possible to verify that the data is read correctly after transfer from one operating system to another.

In any case then the main program initializes `GDF` first by calling GDF$INIT. The value of first argument passed to the routine is set to `'Z'` instructing the routine to initialize ZEBRA. Like all subroutines GDF$INIT returns a status code of `IERR=0` if sucessful. Run time options are set by calling GDF$OPTION, see **??** for details. Before the program terminates GDF$EXIT is called to release all resources held by `GDF`.

Example main program

```
PROGRAM GDF_EXAMPLE
------------------------------------------------------------------------------
-    main program calling example and test subroutines
------------------------------------------------------------------------------

USE GDF

IMPLICIT NONE
INTEGER      IERR                        ! return code
INTEGER      IARGC                       ! external function
CHARACTER(64) DATA,TEMP,TEST             ! file names

CALL GDF$INIT('Z',IERR)                  ! Z=initialise Zebra
CALL GDF$OPTION('AXP',.TRUE.,IERR)       ! set DEC Alpha option

IF (IARGC(IERR).EQ.0) THEN               ! no input file?
CALL GDF$MANUAL                         !   generate manual
ELSE IF (IARGC(IERR).EQ.1) THEN          ! one input file only?
CALL GETARG(1,DATA)                     !   get data file name
CALL GDF$EXAMPLE_1(DATA,IERR)           !   execute GDF example 1
CALL GDF$EXAMPLE_2(DATA,IERR)           !                       2
CALL GDF$EXAMPLE_3(DATA,IERR)           !                       3
ELSE IF (IARGC(IERR).GE.2) THEN          ! two file names?
CALL GETARG(1,TEMP)                     !   test data output file
CALL GETARG(2,TEST)                     !   test data input file
CALL GDF$TEST(10,TEMP,'W',IERR)         !   make test data file
CALL GDF$TEST(10,TEST,'R',IERR)         !   verify test file
```

```
ENDIF

CALL GDF$EXIT(IERR)                          ! terminate GDF
END
```

1.1 A very simple example

The example shown below is a minimalist program. It opens a filem reads first few events and prints out the run number, any event numbers and frame numbers if the information is available. The call to GDF$OPEN opens an input file for later reading. Inside the loop each call to the routine GDF$READ reads all records from the file up to and including the next event record. The call to GDF$CLOSE closes the file again.

After reading an event the data is available to the caller inside as a Fortran90 derived data type. The data types are defined in the Fortran90 module GDF. One way to find out precisely which information is stored one can take a look at the source for the module in `GDF.FOR` . Chapter 3 of this manual presents and explains the contents of the dereived data types in detail. For the moment is is enough to note that for example to hold the run information there is a derived data type `GDF_RUN` which include a field `RUN` such that `GDF_RUN%RUN` is the run number. The 10 m event number is for example `GDF_EV10%EVENT` and the ADC values for a given event are stored in `GDF_EV10%ADC`. To get access to the `GDF` module data a a `USE GDF` statement must be present.

Whenever the contents of a derived data type object are updated during a GDF$READ a value of `NEW=.TRUE.` is set. For all other objects the filed is set to a value of `NEW=.FALSE.` before the routine returns control to the caller. In this example `NEW` is used to only print out new information. In an analysis program a value of `GDF_EV10%NEW=.TRUE.`, which indicates a newly read event with 10 meter data, would for example trigger the 10 meter event reconstruction.

Example: read input file, print one line per record

```
SUBROUTINE GDF$EXAMPLE_1(FILE,IERR)
!------------------------------------------------------------------------------
!   print out a single variable per record read from file
!------------------------------------------------------------------------------

USE GDF                                          ! GDF definitions

IMPLICIT NONE
CHARACTER(*) FILE                                ! input file name
INTEGER      IERR                                ! return code
INTEGER      I                                   ! guess ..

CALL GDF$OPEN(10,FILE,'R',IERR)                  ! R=readonly
DO I=1,5                                         ! first few events
CALL GDF$READ(10,' ',IERR)                      ! read from unit 10
IF (GDF_RUN%NEW ) PRINT*,'Run number',GDF_RUN%RUN    ! run number
IF (GDF_EV10%NEW) PRINT*,'10m event ',GDF_EV10%EVENT ! 10m event number
IF (GDF_EV11%NEW) PRINT*,'11m event ',GDF_EV11%EVENT ! 11m event number
IF (GDF_FR10%NEW) PRINT*,'10m frame ',GDF_FR10%FRAME ! 10m frame number
IF (GDF_FR11%NEW) PRINT*,'11m frame ',GDF_FR11%FRAME ! 11m frame number
ENDDO
CALL GDF$CLOSE(10,IERR)                          ! close file
END
```

1.2 How to link a complete program

To link the example program and any program that uses GDF the Cernlib [**?**] object libraries are needed. They are available for different computer hardware and operating systems including DEC Alphas (both VMS and Unix), Sun Solaris, Intel PCs running Windows NT, and Intel PCs under Linux. The GDF web page, see page i, has links pointing to information on the CERN software.

The completeGDF source code resides in a single Fortran90 file `GDF.FOR` . Compilation of this file will create a module file, for example `gdf.mod`, and an object code file, for example `gdf.obj`. At a larger site it makes sense to put the GDF module file into the default directory for Fortran90 module files, and to put the object code file, converted into a (shared) library file, into the default directory for object libraries.

The details on how to compile and link depend on the operating system. A Unix script would look like this

Unix example

```
set LIB = /usr/local/cernlib/cern/97a/lib
set Cernlib=(-L$LIB -lpacklib -lmathlib -lkernlib)
f90 -o gdf_example gdf_example.for gdf.for $Cernlib
```

The equivalent sequence for VMS would be:

VMS: DCL

```
$ F90 GDF,GDF_EXAMPLE
$ LINK GDF_EXAMPLE, GDF, -
CERN:[pro.lib]MATHLIB/LIB,PHTOOLS/LIB,PACKLIB/LIB,KERNLIB/LIB
```

The GDF web site, see page i, has the GDF source code and the example source code on it. There may also be already compiled modules and the binary code files.

1.3 More examples

The next example is am extend version of the previous example. After reading from the input file the content of the derived data type objects are printed out by calling the routine GDF$PRINT.

Read input file, print contents

```
SUBROUTINE GDF$EXAMPLE_2(FILE,IERR)

!-------------------------------------------------------------------------------
!-    print out contents of all records read from file
!-------------------------------------------------------------------------------

USE GDF                                          ! GDF definitions

IMPLICIT NONE

INTEGER       I,J
INTEGER       IERR
CHARACTER*(*) FILE                               ! file name

CALL GDF$OPEN(10,FILE,'R',IERR)                  ! R=readonly, unit 10

DO I=1,10000                                     ! read a few events

CALL GDF$READ(10,' ',IERR)                      ! read from unit 10
```

```
IF (GDF_RUN%NEW)  CALL GDF$PRINT(GDF_RUN,IERR)

DO J=1,GDF_TELE_MAX
IF (GDF_CCD  (J)%NEW) CALL GDF$PRINT(GDF_CCD  (J),IERR)
IF (GDF_TRACK(J)%NEW) CALL GDF$PRINT(GDF_TRACK(J),IERR)
IF (GDF_HV   (J)%NEW) CALL GDF$PRINT(GDF_HV   (J),IERR)
ENDDO

IF (I.LE.100) THEN
IF (GDF_FR10%NEW) CALL GDF$PRINT(GDF_FR10,IERR)
IF (GDF_EV10%NEW) CALL GDF$PRINT(GDF_EV10,IERR)
IF (GDF_FR11%NEW) CALL GDF$PRINT(GDF_FR11,IERR)
IF (GDF_EV11%NEW) CALL GDF$PRINT(GDF_EV11,IERR)
ENDIF

ENDDO

CALL GDF$CLOSE(10,IERR)                              ! close file

E N D
```

To illustrate how to link GDF with the data analysis code, the next example calculates the sum of ADC signals for a number of events and then displays the distribution as a histogram. The `GDF_EV10.NEW` flag indicates a new set of ADC values while the flag `GDF_FR10.VALID` indicates that the frame information is present which then is used to estimate the pedestals values.

example: create a histogram of total ADC signal

```
SUBROUTINE GDF$EXAMPLE_3(FILE,IERR)

!-------------------------------------------------------------------------------
!    enter sum of total ADC signal per event into histogram
!-------------------------------------------------------------------------------

USE GDF                                              ! GDF definitions

IMPLICIT NONE

CHARACTER*(*) FILE                                   ! file name

REAL          A,P,S,H
INTEGER       I,J,IERR

COMMON /PAWC/ H(999999)

CALL HLIMIT(-999999)                                ! init HBOOK
CALL HBOOK1(100,'ADC signal',50,0.0,1E3,0.0)        ! book histogram

CALL GDF$OPEN(10,FILE,'R',IERR)                     ! open file on unit 10

DO I=1,999                                          ! for a few events
CALL GDF$READ(10,' ',IERR)                         ! read from unit 10
IF (IERR.NE.0) GOTO 999                            ! end of file? error?
IF (GDF_EV10%NEW.AND.GDF_FR10%VALID) THEN          ! new event? pedestals?
```

```
S = 0.0                                   ! reset sum
DO J=1,GDF_EV10%NADC                      ! all ADC channels
A = GDF_EV10%ADC(J)                     ! ADC value
P = GDF_FR10%PED_ADC1(J)                ! pedestal value
S = S + ( A - P )                       ! add up signal
ENDDO                                     ! next ADC
CALL HFILL(100,S,0.0,1.0)                 ! enter total signal
ENDIF
ENDDO                                        ! next event

999 CALL GDF$CLOSE(10,IERR)                     ! close file
CALL HPRINT(100)                          ! show histogram

E N D
```

The next example shows how to read 10 meter and 11 meter data from the same file. Selected stereo events are then written to a new output file. An event is considered a stereo event if both the 10 meter and the 11 meter records were updated during the last read operation. In addition the satellite clock times are compared. In case the times do not match an error message is printed. If the satellite times match the routine to write data is called.

The variable NEW in each record serves two purposes. Like in the previous example the value indicates whether the record was updated during the last read. Before writting any new data into the output file GDF$WRITE evaluates all NEW variables and then only writes the records which have this variable set to true into the output file.

example: select stereo events and store them into a new file

```
PROGRAM GDF_SELECT
------------------------------------------------------------------------------
-    select stereo events and store them in a new file
------------------------------------------------------------------------------

USE GDF

REAL*8 DELTA                                          ! in days
REAL*8 OFFSET                                         ! difference
DATA   OFFSET / 2.2D0 /                               !  GEOS-GPS

CALL GDF$INIT('Z',IERR)
CALL GDF$OPEN(10, 'mixed.fz','R',IERR)                ! input file
CALL GDF$OPEN(20,'stereo.fz','W',IERR)                ! output file

DO WHILE (IERR.EQ.0)                                  ! any error?
CALL GDF$READ(10,' ',IERR)                          ! read event
IF (GDF_EV10%NEW.AND.GDF_EV11%NEW) THEN             ! stereo?
DELTA = GDF_EV10%UTC-GDF_EV11%UTC
IF (ABS(DELTA*86.4D9-OFFSET).LT.1D0) THEN           ! dT < 1 msec?
CALL GDF$WRITE(20,'C',IERR)                       ! output event
ELSE
PRINT*,'No GPS/GEOS match: ',MSEC1,MSEC2
END IF
END IF
END IF
END DO

CALL GDF$CLOSE(10,IERR)
```

```
CALL GDF$CLOSE(20,IERR)
CALL GDF$EXIT(IERR)

END
```

Chapter 2: Subroutine calls

The names of arguments in the specification of subroutine calls follow a simple convention. Any arguments starting with I-N are of type integer, while arguments starting with CH are character variables. Variable names staring with the letter L indicate a Fortran logical. All other arguments are of type real, unless the detailed description of the the subroutine arguments says something else.

As usual in Fortran all variables are passed by reference. Their value remains unchanged unless an * following the name indicates that a variable is used to return a value to the calling routine.

All routines return a status code of IERR=0 if they complete successfully. Unless stated otherwise any other value indicates an error.

2.1 Control

Initialise package

Fortran binding:

```
SUBROUTINE GDF$INIT(CHOPT,IERR)
------------------------------------------------------------------------
prefrobnications
------------------------------------------------------------------------
IMPLICIT NONE
CHARACTER(*) CHOPT                      ! selected options
INTEGER      IERR                       ! return code
```

Action:
Initialize package. This routine must be called before any other call to GDF.

Options:
If Zebra has not already been initialized by some other package such as HBOOK or HIGZ the option CHOPT='Z' must be set. This instructs the routine to call MZEBRA internaly.

Set run time option

Fortran binding:

```
SUBROUTINE GDF$OPTION(CHOPT,VALUE,IERR)

------------------------------------------------------------------------
set run time options
------------------------------------------------------------------------
IMPLICIT NONE

CHARACTER(*)  CHOPT           ! option to set
LOGICAL       VALUE           ! new state, true or false
INTEGER       IERR            ! return code
```

Action:
Set run time options. The default is for all option to be .FALSE..

Options:

CHOPT='ABORT' on entry RETURN from subroutine if IERR.NE.0
CHOPT='AXP' DEC Alpha 16-bit integer array representation. ZEBRA record length in words.
CHOPT='CIO' use C library IO routines
CHOPT='RESET' on entry set IERR=0 if IERR.NE.0
CHOPT='SUN' set options for Sun Sparc Solaris
CHOPT='VERBOSE' print extra information

Terminate

Fortran binding:

```
SUBROUTINE GDF$EXIT(IERR)
------------------------------------------------------------------------

------------------------------------------------------------------------
IMPLICIT NONE
INTEGER       IERR
```

Action:

Last call to terminate all operations. Resources like Fortran units and files are released. However files should normaly be closed by calling the appropriate routine first.

2.2 File operations

Open file for later read or write

Fortran binding:

```
SUBROUTINE GDF$OPEN(UNIT,FILE,CHOPT,IERR)
------------------------------------------------------------------------
open a file
------------------------------------------------------------------------
IMPLICIT NONE

INTEGER      UNIT        ! Fortran unit number
CHARACTER(*) FILE        ! file name
CHARACTER(*) CHOPT       ! setected options
INTEGER      IERR        ! return code
```

Action:

Opens a sequential Zebra file in data exchange format.

On SUNs c-library routines are used to open and read an input file. While no Fortran unit gets assigned to the file the calling program must still provide a unit number which identifies the file in all further calls.

Options:

CHOPT='R' open file in read only mode
CHOPT='W' create a new file, ready for to write new data
CHOPT='X' (default) selects a binary number representation.
CHOPT='A' selects ASCII representation.

An ASCII representation is sometimes desirable when moving files via a network which is not be fully transparent. Transfer of ASCII files via a network will work as long as the program used does translate upper case characters and a few special characters correctly. Tape files should always be in binary since, since the character representation on tape files may differ (eg. ASCII, EBCDIC). The default value is to use a binary number representation.

Read data from file

Fortran binding:

```
SUBROUTINE GDF$READ(UNIT,CHOPT,IERR)
------------------------------------------------------------------------
read Zebra FZ file until next event record
------------------------------------------------------------------------
IMPLICIT NONE
INTEGER       UNIT        ! IO unit number to read from
CHARACTER(*) CHOPT        ! requested options
INTEGER       IERR        ! return code
```

Action:

This routine reads a ZEBRA FZ data file and copies the information up to and including the next event into derived type data structures in memory. The routine also updates other information that is not part of the event but that was found in the data stream while searching for the next event. Examples are tracking information or new run information. The calling program can determine which derived type data structures were updated by checking the `NEW` field in each data structure. A value of `NEW=.TRUE.` indicates that the stucture was updated during the last call to `GDF$READ`. The next call will rest the value to `NEW=.FALSE.` unless there is new data. At the same time the `VALID` flags are set, indicating whether teh information is still valid or not.

Return code:

A return value ot `IERR=1` indicates that the end of file has been reached. The calling program should has to make sure the file will be closed.

Options:

If `CHOPT='C'` the routine verifies that that the checksum of each record is correct, provided a checksum was calculated. If the data file was written on a computer with a different number representation this option can not be used.

Write data into file

Fortran binding:

```
SUBROUTINE GDF$WRITE(UNIT,CHOPT,IERR)
------------------------------------------------------------------------
convert records into Zebra banks and output them into FZ file
------------------------------------------------------------------------
IMPLICIT NONE

INTEGER       UNIT                    ! unit of FZ file
CHARACTER*(*) CHOPT                   ! selected option
INTEGER       IERR                    ! return code
```

Action:

Write all data records which are flagged by `NEW=.TRUE.` into the output ZEBRA FZ file.

Options:
CHOPT='C' requests a checksum calculation.

Close file

Fortran binding

```
SUBROUTINE GDF$CLOSE(UNIT,IERR)
------------------------------------------------------------------------
de-initialize package
------------------------------------------------------------------------
IMPLICIT NONE

INTEGER   UNIT          ! unit number
INTEGER   IERR          ! return code
```

Action:
Closes the currently open file

2.3 Miscellaneous

Generate GDF manual LaTeX files

Fortran binding:

```
SUBROUTINE GDF$MANUAL
------------------------------------------------------------------------
extract documentation from source code
------------------------------------------------------------------------

This routine is a hot entry to win the price for the ugliest,
most uncomprehensible code with most the GOTO statements,
that still works.

------------------------------------------------------------------------
IMPLICIT NONE
```

Action:
Read the GDF.FOR file and use the information in it to generate the GDF manual as a LaTeX file gdf.tex. The main file gdf.tex includes references to other input file which are also generated.

Print content of derived data type structures

Fortran binding: call as subroutine

```
SUBROUTINE GDF$PRINT(DATA,IERR)
```

Action:
Prints data in derived type data structures. GDF$PRINT is defined as a Fortran90 INTERFACE with module procedures for run, event, frame, CCD and tracking data. DATA can be of type GDF_RUN, GDF_EV10, GDF_EV11, GDF_FR10, GDF_FR11, GDF_CCD, or GDF_TRACK.

Write or read a test data file.

Fortran binding:

```
SUBROUTINE GDF$TEST(UNIT,FILE,CHOPT,IERR)
------------------------------------------------------------------------
write or read a test data file.
------------------------------------------------------------------------
IMPLICIT NONE

INTEGER      UNIT          ! logical unit
CHARACTER(*) FILE          ! file name
CHARACTER(*) CHOPT         ! option
INTEGER      IERR          ! return code
```

Action:

Opens a new Zebra file an writes test data into it or reads the a test file to check that the contents are as expected. When reading a file the contents are checked against data stored in memory when writting a test data file.

Options:

`CHOPT='W'` create a new test file
`CHOPT='R'` read file and check contents

Chapter 3: Derived data types and structures

3.1 General considerations

Each design of a data format has to address a few standard questions. How can the data format software change and stay backwards compatible? Should the software try to detect errors and problems in the data? Which information should be recorded? Only the events (images) themsleves? Should other information such as high voltage values or tracking system be included?

- Almost all data structures must have some *time information* in them, in order to correlate them to other data structures and or external information. At event rates of up to 200 Hz the time information does not need to be much more accurate than a millisecond. The modified Julian day including the fraction of day is used as unit of time and stored as a 64-bit real variable. For a number representation with 50 bits to store the mantissa of the real number [1] the least significant bit is equivalent to 3.8 μs for modified Julian days up to 50000.

  In the event data structures higher accuracy time is available. The prcise time is stored in three 32-bit integer variables which contain the number of modified Julian days, the number of seconds since midnight and the number of nanoseconds since the last full second.

- Integer values are defined for *status information* such as the sky quality or the position of trigger bits in the trigger status word. Clearly defined values should allow the analysis software to use this information more reliably. For a list of possible values see appendix A.

- Each Fortran structure contains a *checksum* over the contents of the record. This allows to detect errors corrupting the data.

- In the later data analysis it would be very inconvenient if the inevitable *changes to the data format* were not backwards compatible. Each Fortran structure starts with the version number of the format used when writing the data. To allow the later insertion of more data words the Fortran structures are organized into sectors of similar type data words. When stored in a ZEBRA bank each sector is preceeded by a header word which indicates the type and the length of the sector. The version number and the sector header words together allow backwards compatible changes in the data format. All this is normally of no concern to the calling program, which can assume that a structure has been updated correctly.

3.2 Global data

3.2.1 Run status

This data structure (figure 3.1) contains global information on the status of the whole detector and observer comments. The information is entered into the data stream before the first event is recorded and whenever there is a change during the run. For instance in case of additional observer comments during the run this record appears several times. The run status record is inserted into the data sequential file in between the event records recorded at the time the observer enters the comment.

The character strings containing information on sky quality and trigger conditions are only there for historical reasons. For new data integer numbers describe sky quality and trigger condition (see appendix A).

3.2.2 CCD camera

The CCD cameras on both telescopes are controlled using the same software. For each image taken by the CCD camera the position of the brightest stars is located. After the image analysis the data records for CCD results from both telescopes are in the same format (figure 3.2).

[1] as for example on Sun SparcStations, the Zebra data exchange format itself uses two more bits

run status information

```
------------------------------------------------------------------------
run information
------------------------------------------------------------------------
integer, parameter :: gdf_run_mcl=16000 ! maximum comment length
type gdf_run_t
sequence
integer*4  version                  ! # software version number
integer*4  reserved                 ! # for future use
integer*4  checksum(2)              ! # data part check sum
real   *8  utc                      ! current UTC time          [mjd]
logical*4  status(gdf_tele_max)     ! detector status           [bits]
integer*4  idate                    ! VAX date, local time     [yymmdd]
integer*4  itime                    ! VAX time, local time     [hhmmss]
integer*4  year                     ! Gregorian year, UTC time
integer*4  run                      ! run number
integer*4  type                     ! type of run
integer*4  sky_quality              ! sky quality
integer*4  trig_mode(gdf_tele_max)  ! trigger setup
integer*4  trig_nhig(gdf_tele_max)  ! min tubes above high threhold
integer*4  trig_nlow(gdf_tele_max)  !                 low  threhold
integer*4  clen                     ! actual length of comment   [bytes]
real   *4  sid_length(1)            ! siderial nominal run length [min]
real   *4  sid_cycle                !          nominal cycle time [min]
real   *4  sid_actual               !          actual logged time [min]
real   *4  trig_thrlow(gdf_tele_max)! low  level trigger thresholds  [V]
real   *4  trig_thrhig(gdf_tele_max)! high level trigger thresholds  [V]
real   *8  utc_start                ! nominal UTC start of run     [mjd]
real   *8  utc_end                  ! nominal UTC end   of run     [mjd]
character  file*80                  ! file name
character  observer*80              ! observer names
character  comment*(gdf_run_mcl)    ! any observer comments
logical*4  new                      ! .TRUE. if just read from file
logical*4  valid                    ! .TRUE. if record contents valid
end type gdf_run_t

type(gdf_run_t), target ::  gdf_run
```

Figure 3.1: Data structure for run status information. The array index 1 refers to the 10 meter telescope while 2 refers to the 11 meter telescope.

CCD camera

```
------------------------------------------------------------------------
CCD  camera results
------------------------------------------------------------------------
integer, parameter :: gdf_ccd_nstar_max= 100 ! max number of stars

type gdf_ccd_t
sequence
integer*4  version                   ! # software version number
integer*4  reserved                  ! # for future use
integer*4  checksum(2)               ! # data part check sum
real   *8  utc                       ! UTC time picture [mjd]
integer*4  telescope                 ! telescope identifier
integer*4  nstar                     ! actual number of stars
integer*4  cycle                     ! number of updates sofar
integer*4  interlace                 ! interlace: on=1,off=0
integer*4  bias                      ! pedestal
integer*4  gain                      ! gain
integer*4  noise_range               !
integer*4  exposure                  ! exposure time          [msec]
integer*4  intensity(gdf_ccd_nstar_max) ! intensity
integer*4  status   (gdf_ccd_nstar_max) ! status
real   *4  marker(2)                 ! marker position center tube
real   *4  dark_mean                 ! dark file mean
real   *4  dark_sigma                ! dark file sigma
real   *4  low_mean                  ! mean,  low live pixel
real   *4  low_sigma                 ! sigma, low live pixel
real   *4  noise_threshold           !
real   *4  noise_level               !
real   *4  star(2,gdf_ccd_nstar_max)    ! 1=x,2=y
logical*4  new                       ! .TRUE. if just read from file
logical*4  valid                     ! .TRUE. if contents valid
end type gdf_ccd_t

type(gdf_ccd_t), target :: gdf_ccd(2)
```

Figure 3.2: Data structure for CCD camera. The same structure is used to define two Fortran records. The first one for the 10 meter telescope the second for the 11 meter telescope.

3.2.3 Tracking system

record!tracking

During a run both telecope tracking systems pass information on the source coordinates, the current mode of operation and the actual elevation and azimuth of the telescope to the data acquisition computer.

Since the tracking control systems are similar the same data format (figure 3.3) is used for both telescopes. The observed source is identified by its right ascension and declination in the FK5 J2000 reference frame. A source name is recorded as well, although in general it should not be used to determine which source the telescope pointed at. For off-source runs the source coordinates remain the same as for the on source run. However the variables describing the telescope pointing direction show the actual orientation. An additional variable records the angle between the actual and the nominal orientation. The differences in the coordinates beween on-source and off-source run are stored in a separate set of variables. If the telescopes are inclined towards an assumed common interaction point the height of this point above ground and the computed changes in elevation and azimuth are available.

tracking system information

```
------------------------------------------------------------------------
tracking
------------------------------------------------------------------------
type gdf_track_t
sequence
integer*4    version      ! software version number
integer*4    reserved     ! # for future use
integer*4    checksum(2)  ! # data part check sum
real   *8    utc          ! current UTC time               [mjd]
integer*4    telescope    ! telescope id
integer*4    mode         ! tracking mode
integer*4    cycle        ! number of updates sofar
logical*4    status       ! telescope status       [bit-pattern]
real   *8    rasc_2000    ! source    right ascension, FK5 J2000 [rad]
real   *8    decl_2000    !           declination    , FK5 J2000 [rad]
real   *8    rasc_today   !           right ascension, FK5 today [rad]
real   *8    decl_today   !           declination    , FK5 today [rad]
real   *8    rasc_tele    ! telescope right ascencion, FK5 today [rad]
real   *8    decl_tele    !           declination    , FK5 today [rad]
real   *8    azimuth      !           pointing, +west, north  =0 [rad]
real   *8    elevation    !           pointing, +up  , horizon=0 [rad]
real   *8    deviation    ! angle nominal/actual position        [rad]
real   *8    rasc_offset  ! RA offset for off-source runs        [rad]
real   *8    decl_offset  ! DE offset for off-source runs        [rad]
real   *8    stl          ! local siderial time                 [rad]
real   *8    height       ! height of interaction point        [meter]
real   *8    azi_incl     ! azimuth change for inclination       [rad]
real   *8    ele_incl     ! elevation change for incl.           [rad]
character*80 source       ! source name
logical*4    new          ! .TRUE. if just read from file
logical*4    valid        ! .TRUE. if contents valid
end type gdf_track_t

type(gdf_track_t), target :: gdf_track(2)
```

Figure 3.3: Data structure for both tracking systems.

3.2.4 High voltage

The status of the high voltage is continually monitored during a run. Each time the currents and voltages are measured their values are included in the data stream. Before a run starts the HV software will normally make sure that the high voltage is on. Therefore the first high voltage information record (figure 3.4) will preceed the first event record. To tell whether the measured voltages and currents are acceptable the nominal values, the current requested values and the allowed tolerances are included as well. In addition to the measured values a status value for each channel and a global status flag for the whole high voltage system is recorded as well (appendix A).

High voltage information

```
------------------------------------------------------------------------
high voltage
------------------------------------------------------------------------
integer, parameter :: gdf_hvc_max  = 640 !  max number of HV channels

type gdf_hv_t
sequence
integer*4 version               ! # software id
integer*4 reserved              ! # for future use
integer*4 checksum(2)           ! # data part check sum
real   *8 utc                   ! UTC time, end of measurement
integer*4 telescope             ! telescope identifier
integer*4 mode                  ! current operation mode
integer*4 nch                   ! channels voltages values
integer*4 cycle                 ! read cycle number
integer*2 status  (gdf_hvc_max) ! status of each HV channel  [bits]
real   *4 v_set   (gdf_hvc_max) ! presently set  voltage    [V]
real   *4 v_actual(gdf_hvc_max) ! actual measured voltage   [V]
real   *4 i_supply(gdf_hvc_max) ! HV supply current         [uA]
real   *4 i_anode (gdf_hvc_max) ! measured anode current    [uA]
logical*4 new                   ! .TRUE. if just read from file
logical*4 valid                 ! .TRUE. if contents valid
end type gdf_hv_t

type(gdf_hv_t), target :: gdf_hv(2)

INTEGER, PARAMETER ::             ! status bit positions, LSB=0,MSB=15
GDF_HV_ON = 0,                  !
GDF_HV_XX = 1,
GDF_HV_YY = 2
```

Figure 3.4: Data structure high voltage information

3.3 Event records and frames

Event records and frame records together account for almost all of the recorded data in a file. There is little difference between the 10m and the 11m records, simply because the data acquisition systems are very similiar. The details of the 10m and 11m data records are described further below. The next few paragraphs first discuss some general points which apply to both the 10m and the 11m records.

The *unsigned 16-bit or 32-bit numbers* read from CAMAC system and stored as integers are interpreted as *signed* numbers by Fortran compilers. If there is no better alternative then such numbers are stored as ZEBRA 32-bit patterns and the program calling a GDF routine is expected to correctly decode the information stored as a 32-bit Fortran LOGICAL. However, to avoid problems like integer overflows or wrong signs numbers are mostly reformatted. For example the total livetime is determined by two 32-bit scaler values. After reformatting the livetime is stored as the integer number of seconds since the start of the run and the integer number of nanoseconds.

ADC values and TDC values are stored in 16-bit Fortran `INTEGER` arrays. If the calling program is coded in Fortran77 then care must be taken not to pass these arrays to a subroutine expecting a `REAL` or 32 bit `INTEGER` array. Only the lower 11 bits of ADC value can be non-zero. The other 5 bits are always zero. The highest ADC bit signals an *ADC overflow*. It is left to the calling program to detect[2]this condition and then deal with this situation, for instance by rejecting the whole event.

The event records take up most of the space when the data is written onto a storage media. The physical length of this structure effectivly determines the total space needed. Storing for example 541 ADCs values in 16-bit words requires 1082 bytes, plus one extra 4 byte header word to indicate the data type and the number of words to follow. An event can include up to several hundred more bytes, for example TDC and counter values, or time information. The ZEBRA overhead is typically 48 bytes per event. Data compression utilities such as `compress` or `gzip` reduce the file size by about one half to a third.

At a trigger rate of 200 Hz a 541 pixel camera produces around $1.5\mathrm{kb} \cdot 200\ \mathrm{Hz} = 350\mathrm{kb/s}$ of data. The average data rates can reach the order of $1\mathrm{Gb}$ per hour or $10\mathrm{Gb}$ per night, or $1\mathrm{Tb}$ per year.

The frame and the event records contain variable length arrays. The maximum length is identical to the declared size of an array defined via parameter statements. The number of valid entries depends on the actual number of ADCs and TDC channels which is included in the data structure. Any program using the GDF data structure should use these actual numbers, not the maximum length defined in the parameter statements. Future version of GDF are likely to allocate the memory for variable length arrays dynamicaly.

3.3.1 10 meter frame and 11m frame

The purpose of the 10m and 11m frame records is identical. While the readout system takes events extra data is recorded for calibration and monitoring purposes. An example are the single channel discriminator rates. Before summer 1997 the ADC values taken at a random time which can be used to calculate 'night sky' pedestal values where part of the frame data structure. For new data the number of ADC values in a frame is always zero. To simplify the readout the ADC values for pedestals events are included in normal event records. A trigger bit set in the event record indentifies these events as pedestals events.

3.3.2 10m and 11m event

Figure 3.7 shows the data structure for the 10 meter event information.

For each event this structure (figure 3.8) contains ADC values, TDC values, scaler contents and memory contents read out from the CAMAC system. The number of ADCs and TDCs is part of the record itself since the camera and its geometry might change from year to year. It should still be possible to analyse data from several years together.

Previously the type of event was recorded as an integer number. This scheme does not allow an event to pass several trigger criteria simultaneously. Although not used at present a trigger bit pattern is included inside which any of 32-bits can be set. This allows to record the status of several trigger signals in parallel.

The UTC time information from the GPS or GEOS module is decoded and then stored as `REAL*8` variable containing the modified Julian day, and the fraction of day. The status of the satellite clock module is recorded in the status word.

Pairs of two ADC or TDC values are stored within a 32-bit word. Zebra treats each such word as a bit pattern, which makes sure each bit appears in the same position regardless of the computer used. However, when referring to the 16-bit half word using an `integer*2` array the result can be different depending on the computer used. The array ordering is correct on DecStations and Alphas, while on SUN and other systems with a IEEE number representation the ordering will be wrong. However normaly this is of no concern to the caller. If necessary a internal GDF routine will swap the half words. Still it is a good idea to compare a ADC values against the original values on the data acqusition computer after installing a new version of GDF.

[2]A convenient way to test the overflow bit is the FORTRAN intrinsic logical function `BTEST(ADC,10)`. Note that the least significant bit is at position 0. Position 10 in `BTEST` refers to the 11th bit.

10 meter frame

```
------------------------------------------------------------------------
10 meter frame
------------------------------------------------------------------------
maximum number of adcs and scalers
integer, parameter ::
gdf_fr10_nadc= 636,                  ! max ADCs   (12 per module)
gdf_fr10_nsca= 640,                  !     TDCs   (32 per module)
gdf_fr10_nphs=   8                   !     scaler ( 8 per module)

type gdf_fr10_t
sequence
integer*4 version                ! software version number
integer*4 reserved               ! # for future use
integer*4 checksum(2)            ! # data part check sum
real   *8 utc                    ! current UTC time              [mjd]
logical*4 status                 ! detector status bits         [bits]
logical*4 mark_gps               ! Last recorded GPS (perhaps!) [50ns]
integer*4 nphs                   ! number of phase TDCs
integer*4 nadc                   ! number of ADCs
integer*4 nsca                   ! number of scalers
integer*4 run                    ! run number
integer*4 frame                  ! frame number
integer*4 gps_mjd                ! soon: modified Julian days     [mjd]
integer*4 gps_sec                ! soon: seconds since midnight   [sec]
integer*4 gps_ns                 ! soon: time from last GPS second [ns]
integer*2 cal_adc (gdf_fr10_nadc) ! ADC with internal test voltage
integer*2 ped_adc1(gdf_fr10_nadc) ! ADC first  random event
integer*2 ped_adc2(gdf_fr10_nadc) ! ADC second random event
integer*2 scalc   (gdf_fr10_nsca) ! current monitor scaler
integer*2 scals   (gdf_fr10_nsca) ! single rates scaler
integer*2 gps_clock(3)           ! GPS time last event          [bits]
integer*2 phase_delay            ! phase delay module settings
integer*2 phs1    (gdf_fr10_nphs) ! phase TDC first  random event
integer*2 phs2    (gdf_fr10_nphs) ! phase TDC second random event
integer*2 gps_status(2)          ! soon: GPS status flags
integer*4 align                  ! # 64 bit alignment
logical*4 new                    ! .TRUE. if just read from file
logical*4 valid                  ! .TRUE. if contents valid
end type gdf_fr10_t

type(gdf_fr10_t), target :: gdf_fr10
```

Figure 3.5: Data structure for 10 meter frame.

11 meter frame

```
------------------------------------------------------------------------
11 meter frame
------------------------------------------------------------------------
maximum number of adcs and scalers for 169 pmt channels
integer, parameter ::
gdf_fr11_nadc= 180,                  ! ADCs (12 per module)
gdf_fr11_ntdc= 176,                  ! TDC  ( 8 per module)
gdf_fr11_nphs=   8,                  ! phase TDCs ( 8 per module)
gdf_fr11_nsca= 192                   ! scaler     (32 per module)

type :: gdf_fr11_t
sequence
integer*4 version                ! software version number
integer*4 reserved               ! # for future use
integer*4 checksum(2)            ! # data part check sum
real   *8 utc                    ! UTC time of last event        [mjd]
logical*4 status                 ! detector status flags        [bits]
logical*4 mark_gps               ! Last recorded GPS (perhaps!)  [50ns]
integer*4 nphs                   ! number of phase TDCs
integer*4 ntdc                   ! number of TDCs
integer*4 nadc                   ! number of ADCs
integer*4 nsca                   ! number of scalers
integer*4 run                    ! run number
integer*4 frame                  ! frame number
integer*4 gps_mjd                ! soon: modified Julian days     [mjd]
integer*4 gps_sec                ! soon: seconds since midnight   [sec]
integer*4 gps_ns                 ! soon: time from last GPS second [ns]
integer*4 align_1                ! # 64 bit alignment
integer*2 cal_adc (gdf_fr11_nadc) ! ADC value with internal signal
integer*2 ped_adc1(gdf_fr11_nadc) ! ADC value first  random event
integer*2 ped_adc2(gdf_fr11_nadc) ! ADC value second random event
integer*2 tdc1    (gdf_fr11_ntdc) ! TDC value first  random event
integer*2 tdc2    (gdf_fr11_ntdc) ! TDC value second random event
integer*2 scalc   (gdf_fr11_nsca) ! current monitor scaler
integer*2 scals   (gdf_fr11_nsca) ! single rates scaler
integer*2 geos_clock(3)          ! Geos time last event [bits]
integer*2 phase_delay            ! phase delay module settings
integer*2 gps_status(2)          ! soon: GPS status flags
integer*2 phs1    (gdf_fr11_nphs) ! phase TDC first  random event
integer*2 phs2    (gdf_fr11_nphs) ! phase TDC second random event
integer*4 align_2                ! # 64 bit alignment
logical*4 new                    ! .TRUE. if just read from file
logical*4 valid                  ! .TRUE. if contents valid
end type gdf_fr11_t

type(gdf_fr11_t), target :: gdf_fr11
```

Figure 3.6: Data structure for 11 meter frame.

10 meter event

```
------------------------------------------------------------------------
10 meter event
------------------------------------------------------------------------
maximum number of adcs and scalers
integer, parameter ::
gdf_ev10_nadc = 636,            ! max ADC (12 per module)
gdf_ev10_nphs =   8,            !     phase TDC  ( 8 per module)
gdf_ev10_nbrst=  12,            !     burst TDC
gdf_ev10_ntrg =  65             !     trigger pattern data words

type :: gdf_ev10_t
sequence
integer*4  version               ! software version number
integer*4  reserved              ! # for future use
integer*4  checksum(2)           ! # data part check sum
real   *8  utc                   ! GPS UTC time of event          [mjd]

integer*4  nadc                  ! number of ADCs
integer*4  run                   ! run number
integer*4  event                 ! event number
integer*4  live_sec              ! live time from start of run  [sec]
integer*4  live_ns               ! live time last incomplete sec [ns]
integer*4  frame                 ! frame number
integer*4  frame_event           ! events within frame
integer*4  abort_cnt             ! number of aborts in frame
integer*4  nphs                  ! number of phase TDCs
integer*4  nbrst                 ! number of burst scalers
integer*4  gps_mjd               ! soon: modified Julian days     [mjd]
integer*4  gps_sec               ! soon: seconds since midnight   [sec]
integer*4  gps_ns                ! soon: time from last GPS second [ns]
integer*4  ntrg                  ! number of trigger patterns
integer*4  elapsed_sec           ! sec from start of run          [sec]
integer*4  elapsed_ns            ! ns since last elapsed sec      [ns]
integer*4  grs_clock(3)   ! Wisconsin TrueTime interface [bits]
integer*4  align                 ! 64bit alignment

logical*4  trigger               ! trigger information         [32bit]
logical*4  status                ! detector status flags       [32bit]
logical*4  mark_gps              ! last recorded GPS            [50ns]
logical*4  mark_open             ! last calibration mark open   [50ns]
logical*4  mark_close            ! last calibration mark close  [50ns]
logical*4  gate_open             ! last event gate open         [50ns]
logical*4  gate_close            ! last event gate close        [50ns]

logical*4  pattern(gdf_ev10_ntrg) ! pattern trigger output      [32bit]

integer*2  adc  (gdf_ev10_nadc)   ! event ADC's
integer*2  gps_clock(3)           ! GPS satellite time           [bits]
integer*2  phase_delay            ! phase delay module settings
integer*2  phs  (gdf_ev10_nphs)   ! phase TDCs
integer*2  burst(gdf_ev10_nbrst)  ! burst scalers
integer*2  gps_status(2)          ! soon: GPS status flags
integer*2  track(2)               ! telescope angle encoders     [bits]
logical*4  new                    ! .TRUE. if just read from file
logical*4  valid                  ! .TRUE. if contents valid
end type gdf_ev10_t

type(gdf_ev10_t), target :: gdf_ev10
```

Figure 3.7: Data structure for 10 meter events.

11 meter event

```
------------------------------------------------------------------------
11 meter event
------------------------------------------------------------------------
maximum number of adcs and scalers for 169 pmt channel
integer, parameter ::
gdf_ev11_nadc  = 180,            ! ADC, 12 per module
gdf_ev11_ntdc  = 176,            ! TDC,  8 per module
gdf_ev11_nphs  =   8,            ! phase TDC
gdf_ev11_nbrst =  12             ! burst TDC

type :: gdf_ev11_t
sequence
integer*4  version               ! software version number
integer*4  reserved              ! # for future use
integer*4  checksum(2)           ! # data part check sum
real   *8  utc                   ! UTC time of event            [mjd]
logical*4  trigger               ! trigger bit pattern         [bits]
logical*4  status                ! detector status flags       [bits]
logical*4  mark_gps              ! last one second GPS marker   [50ns]
logical*4  mark_open             ! last opening time of cal mrk [50ns]
logical*4  mark_close            ! last closeing time: Cal time [50ns]
logical*4  gate_open             ! last event gate open         [50ns]
logical*4  gate_close            ! event gate close             [50ns]
integer*4  nbrst                 ! number of burst scalers
integer*4  nphs                  ! number of phase TDCs
integer*4  ntdc                  ! number of TDCs
integer*4  nadc                  ! number of ADCs
integer*4  run                   ! run number
integer*4  event                 ! event number
integer*4  live_sec              ! live time from start of run   [sec]
integer*4  live_ns               ! live time from start of run    [ns]
integer*4  frame                 ! frame number
integer*4  frame_event           ! number of event within frame
integer*4  abort_cnt             ! number of aborts in frame.
integer*4  gps_mjd               ! soon: modified Julian days     [mjd]
integer*4  gps_sec               ! soon: seconds since midnight   [sec]
integer*4  gps_ns                ! soon: time from last GPS second [ns]
integer*4  align                 ! # 64 bit alignment
integer*2  adc  (gdf_ev11_nadc)  ! event ADC's                     [?]
integer*2  tdc  (gdf_ev11_ntdc)  ! event TDC's                     [?]
integer*2  geos_clock(3)         ! GPS satellite time
integer*2  phase_delay           ! phase delay module settings
integer*2  gps_status(2)         ! soon: GPS status flags
integer*2  phs  (gdf_ev11_nphs)  ! phase TDC                       [?]
integer*2  burst(gdf_ev11_nbrst) ! burst scalers
integer*2  track(2)              ! telescope angle encoders     [bits]
logical*4  new                   ! .TRUE. if just read from file
logical*4  valid                 ! .TRUE. if contents valid
end type gdf_ev11_t

type (gdf_ev11_t),target :: gdf_ev11
```

Figure 3.8: Data structure for 11 meter events.

Chapter 4: Implementation

From the viewpoint of the calling program the subroutines provided by GDF simply copy the contents of Fortran structures into files and vice versa. As long as reading and writing is reasonably fast and disk space overheads are small it is of not much interest to know how this actually happens. This is only true as long as there is no need to modify the software and adapt it to changed requirements. This chapter describes some of the basic ideas and techniques which should be kept in mind when planning future changes.

4.1 Fortran records

Data is exchanged with the calling program via data structures specifiea a Fortran90 dreived data types. The reason to choose this method are:

- The previous formats [**?**] and older Fortran77 GDF versions used structures as well. Therefore the it should be easy to adapt the existing analysis software, which represents a large investment and should not become obsolete. Ideally all that is required should be to rename a few variables and subroutines.
- The caller is shielded from the details like number representation, or operating system dependent I/O operations.
- Reference by name is more user friendly (leads to rather more readable, structured and shorter code) and thereby safer than reference by an integer pointer to the location of a data word with an large array.
- To group variables in a structure allows to have multiple instances of the same structure. For example the two seperate records describing the CCD results for both telescopes are identical.

4.2 Choice of data representation software package

Almost every high energy experiment needs to store and then exchange data between different computers. Most experiments choose to adopt an existing software package to solve the problem for them. Instead of trying to come up with a completly new solution the GDF software acts as an interface been the caller who only deals with Fortran records and the Zebra package [**?**]. The advantages of using Zebra are:

- Zebra represents many years of experience in how to store and handle data from high energy physics experiments.
- It has a large worldwide user community. Many high level application programms (PAW) and packages (HBOOK,HIGZ,KUIP) use Zebra to store and exchange data.
- It is supported by the CERN Application Software group, which includes porting it to new computer models (DEC Alpha) and operating systems (Solaris, NT). The use of the software is free and it is available both as an object library or as source code.
- The Zebra-FZ record header allows to quickly search through a large data set to select events because it eliminates the need to read the whole event. Direct access to events records could be impletemented if for some resason seqquential reading of event is not longer sufficient.
- The Zebra-FZ exchange file mechanism provides machine independent reading and writing of data readable on different computers.
- The data can be exchanged over computer networks either in binary format (using FTP or ZFTP) or if the network is not truely transparent in ASCII format.
- Since Zebra banks are self describing future format changes present no problem.
- Zebra includes routines for error detection by checksum and is able to recover the remaining data from files if a physical record block is no longer readable.

4.3 Relation between Zebra banks and Fortran structures

For each Fortran structure a corresponding Zebra bank is defined. The contents of a record are copied into a Zebra bank without any changes. This is stricly true only for new data. To be able to read old data after a changes in the Fortran record format each structure is devided into a header (version number, checksum, UTC time) and sectors. Sectors are blocks of identical type variables (32-bit patterns, integer, real, double precision real, character). The sectors are preceeded by a word which describes the length and the type of sector. In future a sector within a Fortran structure may become larger. It is then still possible to copy banks written in a previous format into the new format sectors because the software is able to tell the difference between bank and structure by looking at the sector header words. If there is any difference it can then introduce empty data words as needed.

Adding completly new Fortran structure does not require much effort. Apart from defining the structure itself only a few arrays have to be increased in length.

4.4 Relation between Zebra records and Zebra store divisions

Zebra organizes the memory space into Zebra divisions. The file space is devided into Zebra event records. GDF uses the convention that the contents of a division end up in the same record inside a file. For instance all event data is collected as a Zebra bank structure inside one division and the written into the same data file record. This record may contain both the 10 meter and the 11 meter information.

Information of validity time goes into different divisions. Whenever data is read from a record/divison with a longer validity span all Fortran records with shorter validity a flaged as invalid. For example after reading a new run status record all other records become invalid.

Appendix A: Parameter values

Predefined parameter values

```
C-------------------------------------------------------------------------------
C     global parameter
C-------------------------------------------------------------------------------
INTEGER, PARAMETER ::
.  GDF_TELE_10 =1,
.  GDF_TELE_11 =2,
.  GDF_TELE_MAX=2
C-------------------------------------------------------------------------------
C     sky quality
C-------------------------------------------------------------------------------
INTEGER, PARAMETER ::
+  SKY_GOOD     =1,      ! definitly good data (A)
+  SKY_UNCERTAIN=2,      ! probably good data  (B)
+  SKY_BAD      =3       ! definitly not good  (C)
C-------------------------------------------------------------------------------
C      trigger mode
C-------------------------------------------------------------------------------
INTEGER, PARAMETER ::
+  TRIG_MODE_SINGLE =1,
+  TRIG_MODE_DUAL   =2

C---- position of trigger bits (LSB=0, MSB=31)
C
C     The old trigger bit position are arranged according to the GRALP
C     trigger numbers. When old GRALP files are converted to this format
C     the trigger bit corresponding to the event code is set.
C

INTEGER, PARAMETER ::
+  GDF_TRIG_SHORT= 1, ! GRALP only, obsolete
+  GDF_TRIG_LONG = 2, ! GRALP only, obsolete
+  GDF_TRIG_TEST1= 3, ! fixed period trigger,  no  TDC time delay
+  GDF_TRIG_TEST2= 4, ! fixed period trigger, with TDC time delay
+  GDF_TRIG_WWVB = 6, ! WWVB time marker
+  GDF_TRIG_STIME= 7, ! siderial time marker, obsolete?
+  GDF_TRIG_HIG  = 8, ! high level trigger
+  GDF_TRIG_LOW  = 9, ! low level trigger
+  GDF_TRIG_EAS  =12  ! GRALP, EAS trigger, obsolete

C----
C     The Hytec based system has its own trigger bits conventions.
C     Note that a given event may have more than one trigger bit set.
C
INTEGER, PARAMETER ::
+  GDF_TRIG_PED = 0, ! pedestal trigger
+  GDF_TRIG_PST = 1, ! pattern selection trigger
+  GDF_TRIG_MUL = 2  ! multiplicity trigger
C-------------------------------------------------------------------------------
C     type of run
C-------------------------------------------------------------------------------
INTEGER, PARAMETER ::
.  GDF_RUN_TYPE_STEREO=1,
.  GDF_RUN_TYPE_10    =2,
.  GDF_RUN_TYPE_11    =3,
.  GDF_RUN_TYPE_MC    =4
C-------------------------------------------------------------------------------
C     telescope tracking status/mode
C-------------------------------------------------------------------------------
INTEGER, PARAMETER ::
.  GDF_TRACK_MODE_ON     =1,
.  GDF_TRACK_MODE_OFF    =2,
.  GDF_TRACK_MODE_SLEWING=3,
```

```
.  GDF_TRACK_MODE_STANDBY=4,
.  GDF_TRACK_MODE_ZENITH =5,
.  GDF_TRACK_MODE_CHECK  =6,      ! pointing check
.  GDF_TRACK_MODE_STOWING=7,      ! stowing telescope
.  GDF_TRACK_MODE_DRIFT  =8,      ! drift scan
.  GDF_TRACK_MODE_MAX    =8

C--------------------------------------------------------------------------
C     position of HV status bits (March 1998)
C--------------------------------------------------------------------------
INTEGER, PARAMETER ::
.  GDF_HV_HWENABLED    =  0,  ! hardware channel enabled
.  GDF_HV_RAMPUP       =  1,  ! output ramping to higher absolute value
.  GDF_HV_RAMPDOWN     =  2,  ! output ramping to lower absolute value
.  GDF_HV_SWENABLE     =  3,  ! software channel enabled
.  GDF_HV_TRIP_SUPPLY  =  4,  ! trip condition: violation of supply limit
.  GDF_HV_TRIP_CURRENT =  5,  ! trip condition: violation of current limit
.  GDF_HV_TRIP_ERROR   =  6,  ! trip condition: voltage error
.  GDF_HV_TRIP_VOLTAGE =  7,  ! trip condition: violation of voltage limit
.  GDF_HV_CRATE_ID     =  8,  ! crate: id
.  GDF_HV_CRATE_STATE  =  9,  ! crate: HV state        [0 off, 1 on]
.  GDF_HV_CRATE_EEPROM = 10,  ! crate: eeprom status   [0 bad, 1 ok]
.  GDF_HV_BATTERY      = 11,  ! crate: battery status  [0 bad, 1 ok]
.  GDF_HV_24V          = 13,  ! crate: 24 V status     [0 bad, 1 ok]
.  GDF_HV_PANIC        = 14   ! crate: panic condition [0 not active, 1 active]
```

Appendix B: Converting old data

A dataset may only be available in a pre GDF data format. The Fortran77 version of GDF included subroutines to convert events from the previous format into the GDF format. If this conversion is still needed then the source code can be taken from earlier version of GDF.

GDF itself tries to be backward compatible and will read files written by earlier version of GDF.

Appendix C: Source code and documentation

The complete GDF source code and the documentation are maintained in a single file named `gdf.for`. The file is available from the web server described on page i.

Any Fortran90 compiler should accept the source code as it is. The code contains no external references to other Fortran90 modules.

The routine GDF$MANUAL read the `gdf.for` file and the creates a set of LaTeX files. The main file is `gdf.tex` which references all the other LaTeX files. After compilation with LaTeX and then `dvips` the file `gdf.ps` contains the complete manual.

Appendix D: Version history and future changes

Version history

```
C
C   0.00 Patchy, Zebra, VMS Fortran records, LaTeX documentation
C   0.01 self describing Zebra sector words
C   0.02 now running both on Sun and under Ultrix
C   0.03 conversion of existing 10m data into this format
C   0.04 c-library i/o for binary files on SUN
C   0.05 several words added to CCD record
C   0.05 modified copy of datman.f no longer needed, removed from card file
C   0.06 structures modified for HYTEC readout, rewriting old stereo data
C   0.07 calibration new, CCD and tracking revised, swap integer*2 on SUNs
C   0.08 optional checksum calculation
C   0.09 valid flags set/reset by GDF$READ
C   0.10 event selection and non-stereo mode in GDF$CONVERT now working
C   0.11 new variable GDF_CAL(i).WIDTH_ERROR(j), more parameters
C   0.12 tracking added in GDF$PRINT, new tracking modes defined
C   0.13 Monte Carlo interface
C   0.14 CCD and HV cycle numbers
C   0.15 UTC time of nominal run start and end of run
C   0.16 long observer comments, GDF$MOVE for shorter than expected records
C   0.17 routines for MOCCA/GDF (Pascal/Fortran) interface
C   0.18 c-library IO while writting binary FZ file on Sun.
C   0.19 routine GDF$RESIZE to change length of last sector at run time
C   0.20 64 bit memory addresses for AXP/OSF-1
C   0.21 coordinate systems for Monte Carlo revised
C   0.22 changes in 10m part for new Hytec Camac readout
C   0.23 all sectors in a structures may increase in length
C   0.24 GDF$SWAP (only used on Sun) revised to swap all integer*2 arrays
C   0.25 GDF_DUMP program to dump file contents on terminal screen
C   0.26 print out of calibration record in gdf$print
C   0.27 64-bit alignment for Alpha under VMS and OSF
C   0.28 changes by Glenn Sembroski to 11m frame and event format (March 95)
C   0.29 file generation by ypatchy and DCL script revised
C   0.30 merged with additional 11m changes to Version 22 by GHS
C   0.31 run time option to set IERR=0 on entry or to return
C   0.32 GDF$ADJUST to set sector length for variable length ZEBRA banks
C   0.33 in 10m frame variable number of scalers and ADCs
C   0.34 in 10m event variable number ADCs
C   0.35 in 11m frame variable number of scalers, ADCs and TDCs
C   0.36 in 11m event variable number ADCs and TDCs
C   0.37 backwards compatibility for fixed length 10m records
C   0.38 revised trigger word bits assignments
C   0.39 UTC time start of year table updated to year 2004
C   0.40 Leeds GPS clock added in frame and event stuctures
C   0.41 increased number of HV channels, variable length record
C   0.42 added 2 times 16 bit in event records for tracking
C   0.43 change from LaTeX to LaTeX2e
C   0.44 manual revised, example programs combined into one program
C   0.45 GDF$PRINT shows bank version numbers
C   0.46 phase delays in frame records
C   0.47 new routine GDF$RUN to move run crecords
C   0.48 new routine GDF$EVENT11 to move 11m event records
C   0.49 new routine GDF$FRAME11 to move 11m frame records
C   0.50 GDF$READ and GDF$WRITE revised to use new data record routines
C   0.51 phase delay added to event records, replaces unused GPS/GEOS 16 bit
C   0.52 GDF$SECTOR removed, as sector descriptors are now set in GDF$MOVE
C   0.53 new GDF$EVENT10 and GDF$FRAME10 routines, GDF$FIXIT removed
C   0.54 obsolete GDF_RUN.CHSKY and GDF_RUN.CHTRIG removed
C   0.55 GDF$TRACK, GDF$CCD routines
C   0.56 true variable length run record, calling UHTOC(.) for ZEBRA conversion
```

```
C   0.57 obsolete routine GDF$ADJUST removed
C   0.58 GDF$MOVE now swaps words in 16 bit arrays
C   0.59 sector descriptor words in removed from Fortran structures
C   0.60 obsolete GDF$RESIZE removed, as variable sectors now handled localy
C   0.61 test routines GDF$TEST to verify data write / read match
C   0.62 new routine GDF$HV to pack/unpack HV data
C   0.63 check file name length and specify READONLY in GDF$OPEN for read access
C   0.64 for old format records on SUN swap 16 bit half-words after UCOPY
C   0.65 bug in GDF$TRACK fixed, 32-bit patterns now only 1 word, was 2 words
C   0.66 record identifier now in ZEBRA bank, GDF$TEST includes tracking
C   0.67 GDF$HV ignores zero length arrays and pre version 66 HV records
C   0.68 increase number of ADC in calibration record to 541
C   0.69 conversion from VMS FORTRAN 77 Patchy into into single FORTRAN90 file
C   0.70 GDF$MANUAL routine to extract documentation
C   0.71 GDF$TEST produces 10m frame records
C   0.72 backwards compatibilty checked for version 67 files
C   0.73 variable length trigger information in 10m event
C   0.74 elapsed time counter, Wisconsin TrueTime clock interface
C   0.75 LaTeX documentation from card file included in GDF.FOR
C   0.76 document GDF$TEST and GDF$MANUAL
C   0.77 fix UNIX read in GDF$MANUAL, tested on Sun Solaris
C   0.78 revised CCD structure
C   0.79 allow zero length sectors for ADC and trigger data in events
C   0.80 ignore ADC and current monitor data fields in 10m frames
C   0.81 revised HV data structure, GDF$PRINT routine for GDF_HV
C   0.82 fix for SUN Solaris F90 32-bit representaion of 16-bit integers
C   0.83 fix for 16-bit half-word swap in version 82, HV status bits added
C
C   - extend GDF$TEST to run HV, CCD, ....
C   - make PRIVATE the default, declare PUBLIC as needed
C   - detection of hardware an operating system type at run time
C
C   - revise MC data structures. Dynamic memory allocation? Linked lists?
C   - write Monte Carlo interface routines GDF$MCE, GDF$MCP
C   - include C structures (Rod Lessard?)
C   - add Monte Carlo example to manual
C   - revise calibration data structures, use RZ direct access file?
C   - verify that checksum calculation works with new bank header format
C   - simple interactive KUIP interface
C   - read/write multiple runs to/from tape or CD
C   - revise strategy for setting valid flags
C   - spell check documentation
C
C
```

Index