CHANT Manual
Revision 2.3

A.Artamonov, J.Brunner, R.Tsenov

April 30, 2004

Contents

Chapter 1

Introduction

The **CH**orus **AN**alysis **T**ool - **CHANT** - is a basic CHORUS off-line data processing program. It is aimed to:

- provide a unified access to the CHORUS experimental and Monte-Carlo data (both, raw and DSTs) stored in ZEBRA FZ-files, and to the data (geometry, calibration constants etc.) stored in database(s).
- serve as a standard CHORUS reconstruction tool, reading raw data and producing different types of DSTs required for various levels of physics analysis
- serve as a framework for reconstruction algorithm development

CHANT can be separated into two independent pieces:

- the **"FRAME"**, ensuring flexible access to various kinds of data and providing a set of utilities to facilitate data handling. Thus, frame can be used as a base for the physics analysis.
- **"reconstruction MODULES"**, comprising subdetector-related or, more global, higher level, experiment related reconstruction algorithms. The output of the modules, combined by the frame in a standard data structures and written to FZ-file, is a CHORUS DST.

The program was designed to be modular as much as possible: frame has no knowledge about modules, modules are (almost) independent of each other. This approach simplifies CHANT maintenance and modules' development, e.g., addition of a new module does not require any changes in the frame.

What is described in this manual, basically, concerns the frame functionality. The module-related documentation, including data structure descriptions, should be found elsewhere.

Chapter 2

Getting started (installation of the code and compilation)

This chapter describes:

- how to access the sources, libraries and executables of the code;
- how to modify "a little bit" the code and obtain user's own executable (most of the users will be satisfied with this);
- how to install a full copy of sources, produce library and executable (for CHANT developer and/or librarian).

2.1 Where is the code

The code is kept via CVS in the repository /choruscvs on chorus-sw.cern.ch cvs server as a separate module in subdirectory *chant* ($CHORUS environment variable points to */afs/cern.ch/chorus/offl/@sy* directory). This subdirectory contains only reconstruction modules, while all frame-related routines and include files are stored in separate cvs subdirectory *chlib*. More about the CVS (Concurrent Version System) as a version management tool can be found in:
http://choruswww.cern.ch/Offline/cvs-transition.html

On a regular basis plain sources are exported into standard Chorus directories under AFS. We have different "levels" of the code: a production version, the "newest" but reasonably reliable version, and a development version, respectively in:
$CHORUS/pro/src/chant
$CHORUS/new/src/chant
$CHORUS/dev/src/chant

(Level "old" exists with an obsolete version as well.)

Chorus environment variable *CHORUS_ROOT*
(by default *CHORUS_ROOT=$CHORUS/pro*)
can be used to (re)set the desired level.

Further on, the directory *$CHORUS_ROOT/src/chant* will be referred as "standard CHANT directory" and its content as "standard CHANT code".

Under the standard CHANT directory there is one subdirectory for each CHANT module, which, in turn, has *src/, comm/* and *doc/* subdirectories for sources (*.F files), *include* files (*.inc) and documentation (if any...).

Library *libchant.a* is compiled from these sources and stored in *$CHORUS_ROOT/lib*. Respective executable is linked and stored in *$CHORUS_ROOT/bin*. For the time being, library and executable are produced on two UNIX platforms only, AIX and Linux.

Note: Library and executable with a debug compilation flag (-g) are produced and stored in the same places, too. However, the corresponding preprocessed Fortran files (e.g. with preprocessor directives *#include ..., #define...* etc. resolved) are not kept, normally. Thereof, one will not see corresponding sources in the debugger.

2.2 How to build one's own CHANT executable with some private modifications

This section is meant for CHANT users, who would need to modify or/and add only few routines to the standard code. It explains how to use the specially designed for this purpose *user_Makefile*, found in the standard CHANT directory. This file must be used only with GNU *make* command,*gmake*.

2.2.1 User's directory structure

The envisaged in the *user_Makefile* structure of user's CHANT directory is very simple and can be changed by redefining the variables in the configuration section of the makefile. These variables are:

```
ROOT_DIR - top directory where everything happens; current directory by default

COMMDIR = $(ROOT_DIR)comm/ - subdirectory for #include files

OBJDIR = $(ROOT_DIR)obj/ - subdirectory for object and dependency files
```

```
BINDIR = $(ROOT_DIR)bin/ - subdirectory for executable files
```

OBJDIR and COMMDIR have to be created by the user. BINDIR will be created during the first linking.

2.2.2 User's code

One can create source (*.F) files and/or copy (or checkout, if working with CVS) some of them from the standard place anywhere in ROOT_DIR and in directory tree beneath. Necessary include files should go in ROOT_DIR, COMMDIR, or $(ROOT_DIR)user/comm only. One may refer to them in *.F files simply as:

```
#include "filename"
```

without any path in front of *filename*.

User **must** provide, i.e., keep among his sources, his own version of *allmod.F* routine. Otherwise, a dummy version from *chlib* will be taken. The safest way to have a clear idea about the used versions of all the user-redefinable routines from a CHANT frame, is to keep all of them among user sources. To do it, just copy recursively entire *user* directory from a standard chant directory to your ROOT_DIR.

2.2.3 Compiling and linking

First step is to set CHORUS_ROOT to the desired level. This will assure linking against the right libraries *libchant.a, libchIO.a* and *libchlib.a.* Then one has to copy *$CHORUS_ROOT/src/chant/user_Makefile* in ROOT_DIR as *Makefile*. Compilation and linking is done by the command:

```
gmake chant
```

To create executable capable to access information stored in Objectivity database, one has to use the command:

```
gmake chantDB
```

Currently, database access is available only at CERN and only on Linux. Before **linking** or **running chantDB**, one **MUST** setup Objectivity environment, executing, in the same shell, the following command:

```
source $CHORUS/dev/src/chorusdb/setup.sh
```

Executable with information for the debugger in it can be created with:

```
gmake chant-g
```

Suppose, a new user will work on development of TRIGGER module and will use CVS. A possible example of his first session may look like this:

```
chlevel dev
cd /afs/cern.ch/chorus/scratch/week/username
export CVSROOT=:kserver:chorus-sw.cern.ch:/choruscvs
cvs co -f chant/user_Makefile
cvs co -f chant/user
cvs co -f chant/trigger
cd chant
mv user_Makefile Makefile
export ROOT_DIR=./
mkdir obj
gmake chant
cp $CHORUS_ROOT/src/chant/doc/run.chant ./
```

Afterwards, user can edit *run.chant* (an example of the shell script with steering cards included, to execute *chant*) and run it.

If user is not going to make publically available modification to *chant*, but, e.g., develops his own analysis routine(s), there is no need, normally, to use CVS. Then, instead of checking out *chant/user* and *chant/user_Makefile* from CVS repository, one can copy them directly from *$CHORUS_ROOT/src/chant*.

```
cp -r $CHORUS_ROOT/src/chant/user
cp $CHORUS_ROOT/src/chant/user_Makefile Makefile
```

If, at some moment, user will need the newest version of any *chant* or *chlib* routine from CVS repository, he can export it from CVS and place. e.g., in *user/src* subdirectory:

```
cd /afs/cern.ch/chorus/scratch/week/username/chant
export CVSROOT=:kserver:chorus-sw.cern.ch:/choruscvs
cvs export -D 7/25/2000 chlib/inout/quepin.F
mv chlib/inout/quepin.F user/src
rm -fr chlib
```

Please note, that this operation requires write permission to the corresponding cvs directory.

If one wants to link *chant* with private versions of some of the CHORUS libraries (chIO, chlib, chant), he can place them in separate directory, e.g. - *mylibs*, and build chant with the command:

```
gmake USERLIBS=-Lmylibs chant
```

If some non-standard private library should be added to the link command (suppose it is libmy.a in the directory *mylibs*), use:

```
gmake USERLIBS="-Lmylibs -lmy" chant
```

There are some caveats here:

The intention is that preprocessed sources are left in the current directory for the later use by the debugger. So far, the compiler's flags for this are set only for AIX and Linux.

When linking, the CHORUS debug libraries *libchant-g.a, libchIO-g.a* and *libchlib-g.a* will be searched for in *$(CHORUS_ROOT)/lib*. **It happens often that, in the dev area, debug and non-debug versions of these libraries are not the same!** This might lead to unwanted results. If this is the case one should change the definition of the variables LOADLIBS_DEBUG and MAINPROG_DEBUG in the makefile.

Note, that one can define *Makefile*'s variables also on the command line, e.g.:

```
gmake chant CHORUS_ROOT=$CHORUS/dev ROOT_DIR=/otherdir/
```

will use */otherdir/* as a directory for your sources etc. and will search for standard libraries in *dev* area. Do not forget the last slash in ROOT_DIR definition!

Execution of gmake will do the following:

- automatic construction of dependencies between *.o, *.F and *include* files through auxiliary *.d files. They will be kept in OBJDIR for further use;
- compilation of *.F files found everywhere under ROOT_DIR. Search for include files is made in current directory, COMMDIR and *$(CHORUS_ROOT)/src/chant/*/comm* in this order;
- linking of these object files and *$(CHORUS_ROOT)/lib/chantmain.o* against *libchant.a*, libchIO.a and *libchlib.a* in *$(CHORUS_ROOT)/lib*. If the name of user's routine coincide with a library one, the user's will take preference;

- storing the executable(s) in BINDIR.

WARNING: If one copies and modifies CHANT include file(s), used by CHANT *.F files, which are not copied in user's CHANT directory structure, these files will not be recompiled and user's modifications of *include* files will not be proliferated to them. **This is error-prone!** One must copy all the files that make use of the modified include file(s) in his/her ROOT_DIR, or beneath. Better, do not modify CHANT *include* files (COMMON blocks). Create your own ones for using them only in your code!

Notes:

- Compilation and linking flags are provided for all Chorus supported (UNIX) platforms: AIX, Linux, HP-UX and Digital UNIX. However, the standard code is tested to compile and run only on AIX and Linux.
- When using AIX4.3, one should use an older archiver from version 4.1 (the new one is not fully compatible with *gmake*). So, *gmake* AR variable is set to AR = *$(CHORUS)/bin/ar41* for this platform.

2.3 Installation and compilation for developer/librarian

This paragraph is meant for CHANT developer/librarian , who would need to install the sources from the CVS repository and to produce CHANT libraries and/or executables either in standard areas or privately. The main tool for this is the GNU makefile, *Makefile*, found in CHANT module under CVS and in the standard CHANT directory (*$CHORUS_ROOT/src/chant*). This file must be used only with GNU *make* command, *gmake*.

Note: *Makefile* uses (includes) auxiliary file, *Makefile.common*. It resides in the same place.

2.3.1 Directory structure

The envisaged directory structure in the Makefile is set by variables in its configuration section. These variables are:

```
ROOT_DIR - top directory where everything happens; current directory by default
SRCDIR = $(ROOT_DIR)src/ - subdirectory for CHANT source files
OBJDIR = $(ROOT_DIR)obj/ - subdirectory for intermediate object files
                           and dependency files
LIBDIR = $(ROOT_DIR)lib/ - subdirectory for libraries
BINDIR = $(ROOT_DIR)bin/ - subdirectory for executable files
```

OBJDIR, LIBDIR and BINDIR will be created during the first compilation and linking. About SRCDIR, see section 2.3.2.

Let us consider for the following, that one wants to install CHANT in directory */afs/cern.ch/user/s/someuser/software*.

2.3.2 Exporting source code from CVS repository

Note: One must be a member of the **v5:cvschant** AFS protection group for that.

If CHANT *Makefile* is available (e.g. in one of Chorus standard areas) then one can install the sources utilising the target "source" in it:

```
gmake source -f $CHORUS_ROOT/src/chant/Makefile \
             RELEASE=chant-XX-XX-XX \
             ROOT_DIR=/afs/cern.ch/user/s/someuser/software/
```

(Do not forget the trailing slash in ROOT_DIR definition!)

or, equivalently:

```
cd /afs/cern.ch/user/s/someuser/software
gmake source -f $CHORUS_ROOT/src/chant/Makefile RELEASE=chant-XX-XX-XX
```

This command will create directory */afs/cern.ch/user/s/someuser/software/src* (if not existent) and export there the CVS module *chant* with its directory tree.

The variable RELEASE controls which version of CHANT is to be exported (e.g. chant-01-00-02). If one does not specify it, the latest version will be exported.

If CHANT makefile is not available, one should export the code manually by:

```
mkdir  /afs/cern.ch/user/s/someuser/software/src
cd  /afs/cern.ch/user/s/someuser/software/src
cvs checkout -r chant-XX-XX-XX chant
```

2.3.3 Building libraries and executables

After having the code exported, or copied to */afs/cern.ch/user/s/someuser/software/src/* one can build libraries *libchant.a* and *libchant-g.a* and executables *chant* and *chant-g*. This can be done in one step by the following commands:

```
cd /afs/cern.ch/user/s/someuser/software/src/chant
gmake bins
```

This command will build libraries *libchant.a* and *libchant-g.a*, and object files *chantmain.o* and *chantmain-g.o* in LIBDIR (=*/afs/cern.ch/user/s/someuser/software/lib*) and link executables *chant* and *chant-g* in BINDIR (=*/afs/cern.ch/user/s/someuser/software/bin*)

There are targets available in the makefile for separate compilation and/or linking, namely:

```
gmake libchant will build libchnat.a
gmake chant       "  "    chant (and rebuild libchant.a if needed)
gmake chantDB     "  "    chant (and rebuild libchant.a if needed)
gmake libchant-g  "  "    libchant-g.a
gmake chant-g     "  "    chant-g (and rebuild libchant-g.a if needed)
```

If one wants preprocessed sources to be left in the current directory for later use by debugger, one should use instead (on AIX):

```
 gmake libchant-g FDEBFLAGS = -d
 gmake chant-g    FDEBFLAGS = -d
```

On Linux, the same is achieved with FDEBFLAGS = -save-temps .

One can *commit* changes of the code to CVS repository by

```
cd /afs/cern.ch/user/s/someuser/software/src
cvs commit chant
```

If necessary, one can tag a version (use "cvs history -T -a" to find out what tags have been created already):

```
cvs rtag chant-XX-XX-XX chant
```

Notes:

- An automatic construction of dependencies between *.o, *.F and include files through auxiliary *.d files will take place before compilation. These files go in OBJDIR for further use.
- Compilation and linking flags are provided for all Chorus supported (UNIX) platforms: AIX, Linux, HP-UX and Digital UNIX. However, the standard code is tested to compile and run only on AIX and Linux.
- When using AIX4.3, one should use an older archiver from version 4.1 (the new one is not fully compatible with *gmake*). So, *gmake* AR variable is set to AR = *$(CHORUS)/bin/ar41* for this platform.

2.3.4 Instructions for CHANT Librarian

- Export the sources to the build area (this gets a "clean" set of sources from the repository, and is hence preferable to using a working directory for ROOT_DIR. See section 2.3.2 for details.
- Build the libraries and executables there (see section 2.3.3)
- Copy the executables, libraries and source to the installation directory. There is target *install* for that in the makefile:

  ```
  cd "working directory"
  gmake  install RELEASE=chant-XX-XX-XX INSTALL_DIR=$CHORUS/dev/
  ```

  (Do not forget the trailing slash in INSTALL_DIR definition!)
- Make symbolic links to the installed files, using target **m**ake_links:

  ```
  gmake  make_links RELEASE=chant-XX-XX-XX INSTALL_DIR=$CHORUS/dev/
  ```

Chapter 3

CHANT frame overview

3.1 Control flow

Very simplified call tree of basic CHANT routines is shown below.

```
CHANT
     PROCIN   main initialization routine
           INIPAC   initialize ZEBRA, HBOOK
           GLOCON   zero global counters, reset flags etc.
           INITIO
           INICOM   read frame related steering cards
           HEADER
           INIFIL
                 CHSLOG   set chIO log level
                 MWOPEN, CHOPEN   open input/output stream(s)
           INIDBS   initialize data base
           INUSER
           ALLMOD   all  modules for global initialisation
           INISUM   print summary of initial conditions
           READIN   initialize read module
           QNEXTE
                 QNEXT
                     ? PROCES   main processing routine
                             READIN   next event readout
                                   QEPCAT   read event catalogue
                                   QUEPIN   read event
                                         CHRDEV
```

```
                  ? EVHEAD
                  ? SPHEAD
                  ? RNHEAD
              IMODUL
        ALLMOD   call reconstruction modules
        USROUT   define user output stream
        PUTOUT   write the output data onto files
              QEPOUT
                    CHWRSE
      ? TERMIN   job termination routine
? TERMIN   job termination routine
```

The main program CHANT is dummy. After initialisation the control is transferred to the routine PROCES. This routine will look after the correct execution of the program, depending on the input data and steering cards. It calls ALLMOD routine of which all user dependent (i.e. ad hoc added code), and all standard reconstruction modules should be hooked. If request for additional ZEBRA memory can not be satisfied for some especially complicated event, the control from ZEBRA is transfered back to QNEXT routine, thus normal program execution is "recovered" by skipping the current event. If the number of recoveries exceeds 100, QNEXT call TERMIN to stop the execution..

3.2 Processing steps

The program can be in different modes, stored in variable IMODE (common /RUNCOM/). Its meanings are:

IMODE = 0	total initialization, banks created here are never dropped.
IMODE = 1	run initialization, banks created here are dropped before next run initialization.
IMODE = 2	event initialisation and processing, banks created here are dropped before next event.
IMODE = 3	run termination, for summaries.
IMODE = 4	job termination, total end of execution.

ALLMOD will be called, first with IMODE = 0, and then 1, 2, or 3, until the program comes to an end, at which time ALLMOD is called with IMODE = 4.

3.3 ZEBRA usage

CHANT stores ZEBRA banks into four different divisions which are wiped out with different frequencies. Therefore the banks booked in these divisions have different lifetimes:

- Type 1: long-lived banks, division IXJOB, never dropped
- Type 2: run-dependent banks, division IXRUN, wiped out at the start of each new run
- Type 3: spill-dependent banks, division IXSPL, wiped out when new spill header event is read in
- Type 4: event-dependent banks, division IXEVT, wiped out at the start of each new event

The division numbers IXJOB, IXRUN, IXSPL and IXEVT are available in the main common /XQSTOR/, therefore it is possible to create banks of different lifetime at the same processing step. Anyway it is recommended to book banks always with division number IXDIV, which is also available in /XQSTOR/ and which is adjusted to one of the three physical divisions depending on the IMODE parameter:

- IXDIV = IXJOB if IMODE=0 (job initialization)
- IXDIV = IXRUN if IMODE=1 (run initialization)
- IXDIV = IXEVT or IXSPL if IMODE>1 (event processing, termination)

Since the I/O is done within framework, normally the user has to worry only about booking, extending or shrinking, dropping, formatting, or printing a bank. He/she should also take care of obtaining (by means of CHANT utilities) and protecting frequently used pointers to the banks against memory relocation due to garbage collection.

For these purposes, only the calls below should be usually used (for the full list of arguments see ”ZEBRA Reference Manual MZ Memory Management”):

Booking:	CALL MZBOOK(IXDIV,...)
Pushing:	CALL MZPUSH(...)
Dropping:	CALL MZDROP(...)
Formatting:	CALL MZFORM(...)
Printing:	CALL DZSHOW(...)
Protecting pointer:	CALL MZLINK/MZLINT(...)

3.4 Frame include files of general interest

The file “main/comm/xqstor.inc” includes files “xqstora.inc” and “xqstorb.inc”. They contain common /XQSTOR/ defining ZEBRA storage. There are many frequently used variables: IXSTOR,IXDIV,IXJOB,IXRUN,IXSPL, IXEVT,LQ(NNQ),IQ(NNQ),Q(NNQ),WS(NWORK),IWS etc... This file should normally be included if there is any access to ZEBRA memory. (Inclusion of xqstora.inc and xqstorb.inc directly in the user program can be useful for a flexible access to ZEBRA working space.)

The file “evnt/comm/evlink.inc” contains common /EVLINK/. There are pointers, filled by the frame, to all the header banks in all ZEBRA divisions as well as to all the 1-st level branches under EVNT:
JEVNT, JRUNH, JJOBH, JSPLH, JGLOB, JRAWH, JUNPK, JMODL, JOTHR.

The file “evnt/comm/rawhlk.inc” contains common /RAWHLK/ with pointers to the raw data banks, filled by frame:
JRSOF, JRESR, JRETU, JRETD, JREAN, JRESC, JREDR, JRETT, JREVT, JREBM, JREMO, JREM1, JRCSC, JRCTU, JRCAU, JRCTB, JRFTR, JRACT, JRACN, JRTBT, JRHDR, JRTDC, JRTCP, JRTC1, JRTRT, JRTDE, JRTBS, JRTLD, JRTMG, JRTXX, JRTER, JRTPA, JEVBI.

“main/comm/implct.inc” contains FORTRAN “IMPLICIT NONE” statement.

The file “main/comm/fractions.inc” contains definitions of possible module’s actions: MDEFLT, MOFF, MFORCE. It should be included if FRACTN function is called.

The file “main/comm/runcom.inc” contains common /RUNCOM/ with banks, flags, and variables for the program execution control. Some of them can be useful in user’s applications:

Variable	description
IMODE	running mode: 0 = global initialisation 1 = run initialisation 2 = normal event processing 3 = run end processing 4 = total program execution end
LABEL	returned from READIN when an ’event’ record has been read, or zero For valid labels, see routine INITLB
IREJEC	reject flag: 0 initially, set to 1 for event rejection
NCEVNT	running event counter
NCRUN	running run counter for runs read on input

NDEVNT	current on-line event number
NDRUN	current on-line run number
IHISFL	histogram flag (0 if no histograms, 1 otherwise)
IVARFL(1)	output flag (0 if no output, ¿ 0 if output: 1 = normal, 2 = copy input banks to output)
IVARFL(7)	machine version (Hollerith)
IVARFL(8)	date of job run (integer)
IVARFL(9)	time of job run (integer)
FVARFL(1)	time (secs.) for time limit recovery (COND 'TIME' t)
XMCINP	.TRUE. if input is simulated data, set at IMODE=1

3.5 Error accounting

Error statistics is accumulated in three parallel banks, ERRC, ERRS and ERRN, created at the job start in IXJOB division. Bank ERRN contains information about error sources, ERRC accumulates total error count for each source, and ERRS stores the last error code for the corresponding source. The frame has no information about reconstruction modules and, thus, about possible error sources. First free slot in the error banks is allocated dynamically at the first error report request for a new error source.

To facilitate the error accounting usage, two interface routines (basically, only the 1-st one is needed) exist. To report an error condition, one calls:

```
CALL ERREPO(module,submodule,icode)
```

If the 1-st argument contains module header bank name followed by three spaces and a single letter denoting severity level, (this form of "call" is strongly recommended) given severity level will be stored in the module header bank and MODL bank. More severe errors will overwrite less severe ones, but not vice versa. E.g.:

```
CALL ERREPO('SPEC   W','SABSFI 5',12345)
```

will store in SPEC and MODL banks "Warning" code (if there was no higher codes stored already) and, as usually, it will insert corresponding information (two char*8 name and error code=12345) in job-live error banks for final error table printout. There are 5 severity levels:

```
Information
Warning         - smt. wrong, but nothing serious
Severe          - module results are useless
```

```
Fatal           - event is useless (request to "call qnext"- skip the
                    rest of event)
Termin          - request to "call termin" (stop the job)
```

There is a possibility to disable "call qnext" and "call termin" for “F” and “T” by means of a new runcard OPTION ERRO, e.g.

```
*DO OPTION -iI
:ERRO n
```

where $n = 1$ to disable "call termin", $n = 2$ to disable "call qnext", and $n = 3$ to disable both.

To get back last error code reported, one calls:

```
CALL ERRGET(module,submodule,icode)
```

First two character*8 arguments identify uniquely error source, e.g.: SPEC,DEDRIF. To get current severity level from MODL or module header bank, the first argument should contain 4-character bank name and "*" in the last position. E.g.

```
character*8  get_code
get_code='SPEC   *'
call (get_code,'        ',ierr)
```

On output, $get\_code(8:8)$ will contain the letter, corresponding to severity level stored in SPEC. Second and third arguments in this form are not used.

Using calls to "errepo", scattered usually over many module routines, one can produce final global error level for module automatically. But, if needed, it can be overwritten by means of *errset* routine, e.g.:

```
call errset('SPEC   F')
```

At the end of the job, the summary error table for all error sources reported is printed out.

The following error report sources are used currently inside CHANT frame:

```
'INOUT   ' 'READIN  '
'MAIN    ' 'IMODUL  '
'I/O    W' 'BAD EVNT'
'I/O    W' 'BAD CATL'
'I/O    W' 'NO E-O-R'
'I/O    W' 'NO S-O-R'
'EDIR - F' 'CHEDSP 1'
'MCDUSE S' 'PATHN NF'
```

Chapter 4

I/O

Centralised, maintained by CHANT framework, I/O, includes different ZEBRA FZ input/output streams and access (only input so far) to HEPDB database. For both cases, there is no direct modules' access to ZEBRA FZ-calls or HEPDB I/O routines. HEPDB information can be extracted by modules through the CHANT utility routine MCDUSE. All the ZEBRA I/O operation are separated into CHORUS general purpose I/O library - chIO (see separate write-up). CHANT frame makes use of chIO to handle FZ-streams and provides "easy-to-use" user callable utility routines for user-type stream handling in modules.

All the data in CHANT, subject to centralised I/O, are ZEBRA data structures.

There is now a possibility to access with CHANT chorus data stored in Objectivity database (ChorusDB). In CHANT memory, these data will be converted into standard ZEBRA bank structure. Two options exist: use ChorusDB as main input stream instead of FZ files, or use FZ file as main input and merge it with ChorusDB data (see description of FILI and OPTION DBPT steering cards).

4.1 Events and labels.

The event consists of the catalogue and the body of event, containing banks. Catalogue contains valuable information allowing for the frame to decide, whether this particular event was requested and, thus, should be read in, or skipped. This information is provided in three words: run number, event number and event LABEL.

When the catalogue of the event has been read from the input unit, CHANT is confronted with one of the following possibilities, as given by the LABEL:

1. LABEL = 201 : run header. The banks will be read. Then, the run initialisation will be performed, and control will be given to the user in ALLMOD with IMODE = 1, if the

run has been selected (input commands CRUN and DRUN). Otherwise, the complete run will be skipped. If the run is not the first run, and there was no run trailer preceding this run header, then this action will be preceded by a proper run termination (see below).

2. LABEL = 202 : run trailer. The banks will be read, the run termination will be performed, and control is given to ALLMOD with IMODE = 3 (but only if this trailer terminates the current run, obviously).
3. LABEL = 203 : spill header event , contains beam and trigger informations
4. LABEL = 204 : calibration event for event builder (not used)
5. LABEL = 205 : calibration event for beam (not used)
6. LABEL = 206 : calibration event for spectrometer
7. LABEL = 207 : calibration event for calorimeter
8. LABEL = 208 : calibration event for optoelectronics
9. LABEL = 209 : calibration event for trigger
10. LABEL = 130 : local event for event builder (not used)
11. LABEL = 131 : local event for beam (not used)
12. LABEL = 132 : local event for spectrometer (cosmics)
13. LABEL = 133 : local event for calorimeter (cosmics)
14. LABEL = 134 : local event for optoelectronics (cosmics, fiducials)
15. LABEL = 135 : local event for trigger (cosmics)
16. LABEL = 111 : "physics" event.
17. LABEL = 115 : local cosmics spectrometer (taken in standalone mode)
18. LABEL = 121 : Monte Carlo event
19. LABEL = 150 : video image

If the event has been selected (steering cards CRUN and DRUN), the banks will be read, and control is given to ALLMOD with IMODE = 2. If the run number in the catalogue bank of the event has changed, and no proper run header and/or run trailer has preceded this event, the corresponding run termination (if appropriate), and run initialisation will be performed as usual. This is meant for catching the accidental loss of run headers and/or trailers. If the

LABEL is unknown (or it was disabled) the event is skipped. Selecting events on the basis of the online-event number (NDEVNT) one has to take into account that all different label types have their own independent event count (apart from 201,202 for which the event number is always 0). Therefore only the combination of the run number, the event number and the label defines uniquely the CHORUS event.

4.2 Labels and ZEBRA divisions.

The LABEL defines also in which ZEBRA division the input data will be stored:

- event label 100 - 199, 207, 208 : storing into event division IXEVT
- event label 203 : storing into spill division IXSPL
- event label 201, 204 - 206, 209 - 299 : storing into run division IXRUN
- event label 202, 300 - 999 : storing into job division IXJOB

Therefore information from events with a label above 300 is valid for the whole job. If there is more than one event of the same kind in one run the older information is replaced by the new one.

4.3 Banks and keys

The following table gives the names and keys for all I/O banks which were supported by the predecessor of CHANT - CHORAL. In CHANT I/O keys of the banks are not used anymore. The only exception is database access where the keys are used as an aliases of the banks. Thus, aliases are used to avoid a boring typing (specifying hepdb pathname to the banks) in many modules. According to the Chorus convention, the 1-st letter of the bank name for the raw data and the database banks means:

R	raw data (DAQ)
D	detector geometry (database)
C	calibration (database)

name	key	origin	comment
RSOF	1001	DAQ	endsystem software related informations
RESR	1011	DAQ	endsystem strobe record
RETU	1012	DAQ	raw data endsystem streamer tubes (dig.)
RETD	1013	DAQ	raw data endsystem streamer tubes (TDC)
REAN	1014	DAQ	raw data endsystem streamer tubes (analog)
RESC	1015	DAQ	raw data endsystem scintillators
REDR	1016	DAQ	raw data endsystem drift chambers
RETT	1017	DAQ	raw data TST planes (TDC)
REVT	1022	DAQ	raw data veto/tracker streamer tubes(dig.)
REBM	1030	DAQ	beam monitoring with FAROS counters
REMO	1080	DAQ	endsystem monitoring (scalers)
REM1	1081	DAQ	endsystem BPUs
EDDC	1090	DAQ	endsystem DAQ data cards
CETD	1093	DAQ	endsystem calibration streamer tubes TDCs
CESC	1094	DAQ	endsystem scintillators calibration
CEPE	1095	DAQ	endsystem scintillators pedestals
CAPE	1096	DAQ	endsystem scintillator calibration
CTTD	1097	DAQ	endsystem TDC calibration
RCSC	1111	DAQ	raw data calorimeter scintillators
RCTU	1112	DAQ	raw data calorimeter streamer tubes (dig.)
RCAU	1113	DAQ	calorimeter run header data
RCTB	1114	DAQ	calorimeter test beam data
RFTR	1211	DAQ	raw data CCDs
RACT	1311	DAQ	CCD pilot old target
RACN	1312	DAQ	CCD pilot new target
RTBT	1510	DAQ	Trigger hit informations
RHDR	1515	DAQ	honeycomb raw data
RTDC	1520	DAQ	Trigger TDC
RTCP	1530	DAQ	Trigger ADC
RTC1	1540	DAQ	Trigger ??
RTRT(*)	1550	DAQ	Trigger burst scalers
RTDE(*)	1560	DAQ	Trigger double event
RTBS(*)	1570	DAQ	Trigger dead time buffered scalers
RTLD(*)	1575	DAQ	Trigger dead time BPUs
RTMG(*)	1580	DAQ	Trigger flash ADC
RTXX(*)	1585	DAQ	Trigger internal
RTER(*)	1590	DAQ	Trigger errors
RTPA	1599	DAQ	Trigger run header
TBCA	1711	DAQ	test beam calorimeter data
EVBI	1901	DAQ	Event builder bank

name	key	origin	comment
MCEV(*)	1999	MC	Monte Carlo informations
RBON(*)	2000	DAQ	beam data (burst related)
RBOU(*)	2001	DAQ	beam data (run start)
JBST	2021	DAQ	version numbers, time stamp
RHCA	2521	DAQ	Calorimeter run header data
QQMN	2601	DAQUAL	data quality
DCSX	3011	DATABASE	x-coord. of center of calorimeter modules
DCSV	3012	DATABASE	v-coord. of center of plane of calorimeter modules
DCSO	3014	DATABASE	orientation of calorimeter modules
DCS1	3015	DATABASE	transverse offset for modules in EM-calorimeter
DCS2	3016	DATABASE	transverse offset for modules in HAD1-calorimeter
DCS3	3017	DATABASE	transverse offset for modules in HAD2-calorimeter
DCSM	3018	DATABASE	readout mask fro calorimeter data
DXPT	3021	DATABASE	x-coord. for wire planes of endsystem streamer tubes
DVPT	3022	DATABASE	v-coord. for center of endsystem streamer planes
DVTP	3023	DATABASE	transverse offset for wires in endsystem streamer tubes
DIOT	3024	DATABASE	wire orientation for endsystem streamer tubes
DTPH	3030	DATABASE	pulse height calibration for CCDs
DTXP	3031	DATABASE	x-coord. of upstrem surface of fiber tracker planes
DTYZ	3032	DATABASE	reference fiber data (fiber tracker plane)
DTCR	3033	DATABASE	contact print data (ribbon shapes)
DTCF	3034	DATABASE	contact print data (fiducial marks)
DTBA	3035	DATABASE	banana corrections for DT planes
DTIO	3036	DATABASE	projection angles of fiber tracker planes
DDTG	3037	DATABASE	diamond tracker geometry
DDWG	3038	DATABASE	diamond tracker geometry
DDTS	3039	DATABASE	diamond tracker geometry (survey data)
DDGE	3040	DATABASE	new simplified diamond tracker geometry
DXPB	3041	DATABASE	x-coord. of endsystem scintillator planes
DVPB	3042	DATABASE	v-coord. of center of endsystem scintillator planes
DVBP	3043	DATABASE	transverse offset for modules of endsystem sc. planes
DIOB	3044	DATABASE	orientation of endsystem scintillator planes
DXPD	3051	DATABASE	x-coord. of wireplanes of endsystem driftchambers
DVPD	3052	DATABASE	v-coord. of center of endsystem driftchambers
DVDP	3053	DATABASE	transverse offset for wire of endsystem driftchambers
DIOD	3054	DATABASE	orientation of endsystem driftchambers
DVPA	3061	DATABASE	v-coord. of center of endsystem streamer tubes (analog)
DVAP	3062	DATABASE	transverse offset for analog strips of streamer tubes
DMAG	3063	DATABASE	endsystem magnet current parametrisation
DCTX	3091	DATABASE	x-coord. of wireplanes of calo streamer tubes
DCTV	3092	DATABASE	v-coord. of center of calo streamer tubes
DCTO	3094	DATABASE	orientation of calo streamer tubes
DCT1	3095	DATABASE	transverse offset for wires in first plane (EM)
DCT2	3096	DATABASE	transverse offset for wires in second plane (EM)
DCT3	3097	DATABASE	transverse offset for wires in HAD1, HAD2

name	key	origin	comment
CCSR	5011	DB	high/low ratios for calorimeter ADC's
CCSA	5012	DB	attenuation lengths for calorimeter modules
CCSC	5013	DB	equalisation constants for calorimeter modules
CCSP	5014	DB	reference pedestals for calorimeter ADC's
CCND	5015		noisy/dead channels in calorimeter
CCPA	5017	DB	calorimeter parameters
CTDI	5211	DB/UNDIST	distortion correction parameters (OPTO chains)
FUCA	5299	DB	CCD calibration
CESA	5111	DB	endsystem scin. calibr. (att.length)
CESQ	5112	DB	endsystem scin. calibr. (equal. const.)
CESH	5113	DB	endsystem scin. calibr. (high voltage)
CESC	5114	DB	endsystem scin. calibr. (???)
CXTT	5511	DB	trigger TDC (zero time)

If the raw event is read in, the frame stores pointers to the raw data banks in the MZLINK-protected common block. The pointer names are simply composed by a J preceding the bank name. The common block can be used in any user subroutine if it includes the file evnt/comm/rawhlk.inc.

All banks produced by the DAQ contain a header of the following structure:

word	description
1	total bank length (header+data)
2	header length (>10)
3	bank key (see table below)
4	program version (positive integer)
5	Event tag
6	bank status (error word)
7	date (ddmmyy)
8	time (hhmmss)
9	checksum (for online use)
10	trigger bit pattern
11	reserved

Detailed description of the content of these banks can be found in various post-script files (produced by the DZDOC package) at the following places (as part of CHORAL documentation):

- under AFS in directory *$CHORUS/*/doc/ps*
- on the Web:*http://choruswww.cern.ch/Offline/Choral*

Therefore an "event" has the following structure **(THIS IS NOT RELEVANT FOR CHANT EVENT FORMAT !!!)**:

logical record	User header	length of user header	start-of-event flag	bank content
1	CATALOGUE	NKEY+3+(NTRIG+1)(*)	1	non
2	KEY1	1	0	"KEY1"
⋮	⋮	⋮	⋮	⋮
NKEY+1	KEYn	1	0	"KEYn"
NKEY+2	-1	1	0	non

Explanation of terms:

CATALOGUE event catalogue of following content:

word		content
1	-	LABEL
2	-	KEY1
⋮		
NKEY+1	-	KEYn
⋮		
NKEY+1+(NTRIG)-1	-	TRIGBT(*)
NKEY+1+(NTRIG)	-	READPT(*)
NKEY+1+(NTRIG+1)	-	-NTRIG(*)
NKEY+2+(NTRIG+1)	-	-NEVENT
NKEY+3+(NTRIG+1)	-	-NRUN

(*) means optional information, may be omitted

KEY1...KEYn NKEY unique key numbers defining the banks which are following, for raw data NKEY=1 , KEY1=1000, the full key list is filled at the first processing cycle

NKEY number of keys (banks) which are following

TRIGBT one word, containing the trigger bit pattern, only present if NTRG=2 (normally only for LABEL 111 only after the first processing cycle)

READPT readout pattern , only present if NTRG=1 or NTRG=2

NTRG number of trigger words, has to be omitted if NTRG = 0

NTRG = 0	LABEL > 199
NTRG = 1	LABEL < 199 , raw data
NTRG = 2	LABEL < 199 , after first processing cycle

NEVENT online event number (NEVENT = 0 for LABEL > 199)

NRUN online run number (NRUN = 0 for LABEL > 299)

"KEYn" bank identified by KEYn

As it is clear from the above table an event with NKEY banks consists of NKEY+2 logical records. The header and trailer records don't contain banks. The trailer record is there to increase the robustness of the structure because it may happen that some banks which are indicated in the header are missing.

4.4 CHANT standard data structures and related utilities

Unlike the raw data, CHORAL or EFICASS produced data, CHANT ZEBRA banks destined for centralized usage or I/O, are combined into single structure, supported by the top header bank. Depending on the ZEBRA division, the header bank names are: EVNT, RUNH, SPLH (there is no header bank in IXJOB division). In the same time, modules are free to create standalone banks for temporary usage during the event processing. These banks MUST be removed at the module exit.

The most complicated, EVNT, bank structure is organized in several branches: RAWH, UNPK, MODL, MCEV, GLOB and OTHR (detailed description - elsewhere). Each module creates its own top header bank under MODL branch (for reconstruction output) and under UNPK (for unpacked data).

Unlike CHORAL, where banks were identified by "I/O keys", in CHANT the only unique identifier of the bank is its four-character name.

There is a set of utilities facilitating handling ZEBRA data structures in CHANT.

To obtain the number of data words of a bank the integer function

```
LENGTH(JPOINT)
```

is available where JPOINT contains the bank address and similarly

```
NLINKS(JPOINT)
```

supplies the number of structural links of the bank at JPOINT.

`SUBROUTINE EVFANO(JLINK,KL,CHID,NL,BNAME,IVER)`

books module fan-out bank

`IGETBT(NAME1,NAME2,NAME3)`

returns pointer (if positive) to the bank "EVNT"$->$"NAME1"$->$"NAME2"$->$"NAME3". "NAME2" and "NAME3" can be empty.

`IGETBN(JP,NAME)`

returns pointer to the bank "NAME". "JP" is a pointer to the mother bank.

`IGETBL(JP,NAME)`

returns link offset of the bank "NAME". "JP" is a pointer to the mother bank.

`SUBROUTINE GETBNP(JP,NAME,JPD,KL)`

returns pointer ("JPD") and link offset ("KL") to the bank "NAME". "JP" is a pointer to the mother bank.

`SUBROUTINE EVLOOK(CHID,BNAME,KL,JP)`

Look if the CHID fan-out exists in the BNAME Branch. If yes, return pointer ("JP") and link offset ("KL").

4.5 HEPDB data base access: MCDUSE

CHANT has access to an HEPDB database chorus.dbs which contains the geometry and calibration information. HEPDB is based on RZ-ZEBRA files and processing is carried out on an HEPDB server. Data are available in a RZ-tree structure like '//CDCH/GEOMETRY/CALOR/X-COORD'.

The database can be accessed interactively. In CHANT it should be accessed by means MCDUSE interface routine:
CALL MCDUSE(PATHN, CHBANK , LBK, CHOPT, IRC)
where PATH is a character string containing full path to the bank in HEPDB or its alias, preceded by "%" (historically, aliases are the character representation of bank I/O keys, used in CHORAL, e.g., full path '//CDCH/CALIBR/TUBES/X-COORD' can be accessed via alias: '%3021').

CHBANK is a character*4 bank name. This name will be assigned to the bank read from HEPDB, allowing, thus, its further access by name (see table of CHORUS banks above).

In LBK the pointer to the requested bank is returned. MCDUSE must be called by module at each start of run ($IMODE = 1$). MCDUSE can be called (e.g. from different modules) several times per run for the same bank. In the case of repeated call, it looks for the bank with corresponding name in RUNH/GEOM or RUNH/CALI branch and returns its pointer.

Character parameter CHOPT should contain either 'G' or 'C' to tell MCDUSE to/in which RUNH branch, GEOM or CALI, the bank should be attached/looked for. In the case of real data, MCDUSE at first looks for the bank in steering cards, than in HEPDB, and eventually - in RUNH/GEOM or CALI, which should contain bank instances, read in from the (new) dst. If the bank is still missing, it tries to find it directly under RUNH, what can be relevant for the raw data or "old", CHORAL, dst. In the case of Monte-Carlo events, the input stream (i.e. data in run division) takes preference over HEPDB. This logic may be set for real data as well, if CHOPT contains character 'R'.

In case of Monte-Carlo data with run number less than 50000 (such a mistake has been made in many private MC productions), HEPDB is accessed with run number equal to 50000.

If the bank was not found by some reason, nonzero error code is returned in IRC and LBK input value is preserved intact.

MCDUSE prints error/warning/debug messages for "FRAM" print flag at levels from 0 to 5. To suppress any printout, FRAM should be set to "-1", e.g.:
DO PRINT -iI
:FRAM -1

4.6 ASCII I/O: FRDBOUT

There is an utility routine FRDBOUT which, in cooperation with special Perl script, can save single ZEBRA bank (not an entire tree or linear structure !) in a format, suitable for its further input via TZ cards. This possibility, though not realized in a quite natural way, allows to run with a new calibration/alignment constants without entering them in HEPDB. **To be written ...**

Chapter 5

Steering cards

5.1 General remarks

To access steering cards in CHANT standard ZEBRA TZ package is used. According to TZ syntax, the “card” begins with header line followed by data line(s) (for details of the syntax, see ZEBRA manual):

```
*DO <card_name> [<options>]  #. comment
dataline 1
  ...
dataline n
```

The ″*card_name*″ in the header line (only first 4 characters are significant) defines the Hollerith id of the bank, created by TZ, which contains all the data from datalines. TZ routine

```
CALL TZFIND(IXJOB,JP,card_name,0,0)
```

returns in JP pointer to this bank (″*card_name*″ here should be a string of exactly 4 characters long). Comment lines start with “*.” in the first two positions, in-line comments begin with “#.”. Special TZ control line
*PRINT
allows to echo cards on stdout. The list of TZ commands should be completed with
*FINI
command. The character strings in datalines should be preceded by : or be enclosed in ” ” or ’ ’.

CHANT framework recognises only the cards names described in the next section. Module-specific cards should be accessed (using TZFIND) in the module itself.

5.2 Steering cards supported by CHANT frame

Summary of cards and their default values

card name	type of data	default
CRUN	$< ranges >$	1
CRUN	:RUN r1 r2 ... :EVNT n1 n2 ...	none
DRUN	$< ranges >$	1 -999999
DRUN	:RUN r1 r2 ... :EVNT n1 n2 ...	none
CEVENT	$< ranges >$	1
DEVENT	$< ranges >$	1 -999999
CONDITION	:cnd1 n1 ...	none
LABEL	:lab1 n1 :lab2 n2 ...	default list
MASK	BITPAT1 BITPAT2 LABEL ...	none
MODU	:mod1 act1 :mod2 act2 ...	:ALL :DEFA
FILI	:input_file_name ...	none
FILO	:stream_type :output_file_name ...	none
USTR	:key1 n1 ...	none
TITLE	:string	none
DEBUG	$< ranges >$	none
PRINT	:key1 n1 ...	no key
OPTION	:key1 n1 ...	no key
HISTO	$< ranges >$	none

5.2.1 Input data selection

CRUN There are two forms of this command. If it contains only ranges, its format is the same as that of CEVENT, but for run numbers. If it contains the sub-keys 'RUN' and 'EVNT', than for all runs in the first lot following 'RUN', the events following the 'EVNT' will be processed, and so on, any number of 'RUN' followed by 'EVNT' ranges. The CEVENT command must still be present and cover the event ranges requested. Example:

```
*DO CRUN -iI
:RUN 1 :EVNT 157 :RUN 2 :EVNT 1001 -1100
*DO CEVEnt -iI
1 -1100
```

will process event 157 of the first run encountered, and events 1 to 100 of the second run, if the first run contains 1000 events.

DRUN selects on-line assigned DATA taking RUN numbers. (see CRUN)

CEVENT counts the number of events encountered on input, starting at number 1.

DEVENT selects on-line data taking event numbers

LABEL Defines the labels to be excluded from processing, e.g.:

```
*DO LABEL -iI
111 0 199 1
```

will delete (i.e. prevent from being read and processed !) events label 111, and will add a new event label 199. The command

```
*DO LABEL -iI
999 0
```

will delete all labels from the default list. So the combination

```
*DO LABEL -iI
121 1 999 0
```

will only read and process events with label 121.

MASK Defines bit pattern selections for reading from **event directories**. It accepts groups of three numbers: patter1, pattern2 and event label, for which the selections is valid. Patterns can be specified not only in decimal form, but in octal (preceded by #O) or hexadecimal (preceded by #X). The event will be accepted if the following expression is "TRUE": $``pattern2'' = (``patter1''\&``edir_pattern'')$. Example:

```
*DO MASK -iI
#xf #x9 111
```

all the events with label 111 and 1001 binary bit pattern in four LSB will be selected.

OPTION 'TRIG' select events according to their trigger bit pattern (see more details below).

OPTION 'DBPT' provide bit pattern (decimal format) when requesting event from ChorusDB Objectivity/TM database.

bit 0 "on" (integer 1) Choral data (STEV bank) is requested;

bit 1 "on" (integer 2) - Chant data (EVNT bank);

bit 3 (integer 8) - NetScan analysis 4 (reserved);

bit 4 (integer 16) - NetScan analysis 3 (reserved);

bit 5 (integer 32) - NetScan analysis 2 (reserved);

bit 6 (integer 64) - NetScan analysis 1 (Charm, ANEV bank);

bit 7 (integer 128) - reconstructed NetScan data (Nagoya reconstruction, NGEV bank);

bit 9 (integer 512) - EMulsion FeedBack (EMFB bank) data;

So, if Choral(STEV), Chant(EVNT) and reconstructed NS emulsion data are to be read, one has to specify DBPT option 1+2+128 = 131.

STEV and emulsion banks will be attached to the branch OTHR of chant EVNT structure. EVNT tree either will be build by CHANT or obtained, if requested and available, from DB.

In case of FZ (not CHORUSDB) input stream, non-zero value of DBPT requests for each input run/event corresponding information from DB. All the banks obtained from DB, including EVNT, will be attached to OTHR branch, while main EVNT structure will be the one, read from input FZ-file. In such a way one can merge information from FZ DSTs and database.

5.2.2 I/O stream control cards

FILI defines file names for input stream(s). Number of input files is not limited in CHANT, but there is a TZ package limitation of about 10 kb of input data per card. To overcome it, one can specify several FILI cards. Example of FILI card:

```
*DO FILI -AW    #. input file(s)
:mydata/test1.fz
"chstagein -s 1500 Y20469.1"
:‘chstagein Y20412.245‘
:/afs/cern.ch/chorus/edir/1996.16-Jun-96/dst1/Y10094.7.edir
:USER :data/junk/myuser.fz
```

The name of the file can be either UNIX file name, or filename in the staging pool (e.g. /shift/shd14/data21/v5/stage/Y20464.1.al), or chstagein command enclosed in " " (without preceding :), or (ch)stagein command in ‘ ‘. Be aware that in the latter case the execution starts only when all the commands within ‘ ‘ will be executed by the operating system. i.e. all the tapes staged.

Amount of text under single FILI card is limited in TZ-package by 10kb, but FILI card can be repeated many times. Using UNIX means, one can insert under FILI card file or tape names listed in disk text file or create such a listing in-fly. For example:

```
*DO FILI -AW     #. take input file names from text file or ls command
‘awk ’{print "\"chstagein "$0"\""}’ list_of_tapes.txt‘
‘awk ’{print ":"$0}’ list_of_fz-files.txt‘
‘ls -1 /mydir/summ*.fz | cat -n | awk ’($1>=1&&$1<101)’ | awk ’{print ":"$2}’‘
```

If the name of the disk file (or name of the link to the file) has extension “.edir”, the file will be treated as special “EVENT DIRECTORY” file. Such a file should be produced for each CHANT MAXI-DST rz-file. CHANT reads in edir-file and opens (by means of stagein) corresponding fz-file. edir-file contains very compressed information (a few words per event) about each event, including physics summary and event position in fz-file. The latter allows fast direct access to the specified event(s) avoiding sequential scan through all preceding events on the fz-file. The edir bit pattern should provide means for fast selection and analysis of small subset of emulsion data.

If keyword “NONE” is provided instead of input file name, CHANT will simulate dummy event loop without reading file. This can be useful, e.g., if CHANT FRAME is used to produce Monte-Carlo events. Events will have standard EVNT-based ZEBRA structure. Its branches, e.g. MCEV, can be used to attach user banks. Events will have standard catalogue: LABEL, -EVNUM, -RUNNUM. Run number will be taken from “DRUN” steering card (if missing, default value 100001 will be used). “DEVENT” card defines the range of event numbers (and, thus, number of events) to produce. The first label specified in the LABEL card defines labels of generated events.

There is also possibility to generate sequence of run/event numbers, predefined by user in /U_SELEC/ common block. The first variable in this common contains number of events in the list. It is followed by run/event number pairs. Ranges in DRUN and DEVE cards should be wide enough to contain all run/event numbers from the list. This mode can be usefull, e.g., to make asciifile-driven event loop when loading manual scan data into Objy (is now implemented in chant/emfb).

If keyword “CHORUSDB” is provided as input file name, CHANT will retrieve events from Objectivity database. “DRUN” and “DEVENT” steering cards define run and event ranges, the OPTIon card “DBPT” defines type of information requested. If FZ-file is specified as input, non-zero value of “DBPT” will request for each input event corresponding information from database (see description of “DBPT” OPTION).

Keyword ”USER” defines a special user controlled input stream. This stream can be accessed from user the code by means of routine:

```
CALL GETUSR(JP,inlabel,ierr)
```

It returns (if positive) a ZEBRA structure of the next event in the user input stream, the label of this event and the ZEBRA error code. If ZEBRA EOF is reached (code $\geq$ 5), the stream will be automatically closed and disabled for further I/O requests with a corresponding warning message printed. The event will be read in into default ZEBRA division (=0). This stream can be useful, e.g., to read in the calibration banks from FZ-file instead of requesting them from the database.

If the input data are in a new CHANT format (single ZEBRA data structure supported by EVNT, SPLH or RUNH banks), they are copied into memory without modifications. If it is not the case, before calling ALLMOD routine they will be converted into a standard CHANT format (for more details see comments on the output COPY stream below).

FILO defines output streams. There can be up to four output streams: MAXI, MINI, COPY and USER. E.g.:

```
*DO FILO -AW     #. output file(s)
:MAXI :./out1.fz :MINI :./out2.fz :USER :./out3.fz
:COPY :./out4.fz
```

The set of banks (by their names) which should NOT go to the particular stream (except of COPY), can be specified by calling CUTBNK routine, e.g.:

```
        PARAMETER(NMINIBNKS = 2)
        CHARACTER*4 MINIBNKS(NMINIBNKS)
        DATA MINIBNKS/'UNSP','SPEC'/
*
        CALL CUTBNK('MINI',MINIBNKS,NMINIBNKS)
```

directs to MINI stream entire EVNT structure, but UNSP and SPEC branches. Such specifications are being accumulated if called from different modules. If no single call to CUTBNK was made for any particular stream, by default, the entire EVNT structure will be written out.

USTR user output stream control.
In addition to CUTBNK routine, for USER stream, there is a possibility of control via card USTR. E.g.:

```
*DO USTR -iI        #. USER stream control
:MAXI 1 :MODU 0 :UNSP 1
```

takes as a base for user stream definition of MAXI, adds UNSP branch to it and drops MODU branch. By default (if no MAXI or MINI keys are present), the entire EVNT structure is taken as a base. USTR card can exclude also top level banks (EVNT, SPLH, RUNH) to prevent writing out of this structures at all. If keyword :ALL 0 is specified, an automatic writing to user stream is fully disabled.

The routine PUTUSR(JP) can be used to write a ZEBRA structure supported by JP to the user stream. It creates the event with a current label on the user output stream. This feature can be helpful to create an FZ-file containing a single (or some) calibration bank(s) only.

COPY stream performs a “one-to-one” copy of the input data. However, if the old, CHORAL-like, format data are read in (i.e. separate ZEBRA structures, e.g. CHORUS raw data), the raw top banks will be attached to EVNT/RAWH branch, MCEV will be attached to EVNT, and all the rest will go to EVNT/OTHR branch. Similarly, for the event labels corresponding to the spill and the run divisions, the banks will be attached to SPLH or RUNH [1]. Thus, the COPY stream always contains a new, event structure, supported by a single top level bank.

OPTION 'NOFI' By default, when CHANT encounters non-existing input file, it considers this error as fatal and stops execution. To ignore such an errors and continue processing from the next existing file, one can specify 'NOFI' option 2.

CONDITION Various special conditions to be set. Each condition is followed by exactly one number, integer or floating, depending on the condition. The current list comprises:

- **'CERR' n** will skip n I/O errors before ending the job, except for errors 1 (EOF) and 3 (end of information). It is useful in case of corrupted input data. Usually the input file is closed after reaching the EoF mark (system EoF as well as ZEBRA EoF). To skip EoF marks the user may modify the logical function USREOF and set their value to .true.
- **'TIME' t** (floating) will stop event reading t seconds before the end of the job, to allow the end-of-job processing take place (histograms, etc.). t is given in real seconds. The default is 2.

DROP (not yet implemented)

5.2.3 Printout, histogramming, module steering and option control

TITLE allows title to be printed in big characters after the steering cards.

[1] RSUM bank is ignored on input

DEBUG activates PRINT flags for the selected (see CEVENT) counted events. NOTE: another way of activating the print flags, given on a PRINT command, is by calling routine SDEBUG. CALL SDEBUG(1) will activate the print flags, CALL SDEBUG(0) will deactivate them (the both of the actions for the current event only).

MODULE defines the module behaviour providing it with one of the standard "action" flags:

```
DEFA - module default action, i.e. do nothing if output bank(s) exist.
OFF  - do nothing
FORC - drop output bank(s), if present, and reprocess from raw data.
```

Input data in MODU are 4-character pairs of module_key/action. For example:

```
*DO MODU -iI     #. module action, possible keys OFF DEFA FORC
:ALL  :DEFA
:SPEC :FORC
:HONC :OFF
```

"ALL " is a "pseudo module" which sets the default for all other modules. Currently available module keys are: CALO, FTRA, FUJI, HONC, HXTR, MIDI, SPEC, TCAL, TRIG. Module should check the requested action, and proceed accordingly, by means of the frame integer function FRACTN, e.g.:
IACT=FRACTN('SPEC',IERR)
It should be called once at $IMODE = 0$ and its return value should be stored in a saved integer variable. The routine, which calls FRACTN, should include the file "fractions.inc", containing the definitions of possible actions, i.e.: MDEFLT , MOFF , MFORCE.

HISTO sets a flag IHISFL (common /RUNCOM/) to 1 (default 0) indicating that histograms are requested. This flag can be conveniently tested by the users. The ranges can be inquired by using the logical function HISTON to know whether a histogram NUMBER is in the ranges or not:

```
      IF(HISTON(NUMBER)) THEN
```

(don't forget to declare HISTON type LOGICAL). This check should be stored in some faster access form for very tight histogram filling calls. The ranges are there for a user convenience only. All histograms booked will be printed automatically if IHISFL is set, i.e. if a HISTO command has been read. For printing this, the following rule holds:

Histograms with MOD(NUMBER,1000) LE 500 are printed and reset at every run end, all the others are only printed at the end of the job.

PRINT has a special format allowing the use of key words. The key words must be four characters long (not less, not more). The key words are free, the user may add (almost) any number at any time. Format:

```
*DO PRINT -iI
:KEY1 n1 :KEY2 n2 :KEY3 n3  ...
```

There are two functions to obtain these flags, KAPRNT and KEPRNT. KAPRNT will return the print flag at all times, KEPRNT only for events selected by a DEBUG command, otherwise KEPRNT will return zero. Applied to above example,

```
N = KAPRNT('KEY1')
```

will set N to n1, etc.

```
N = KEPRNT('KEY1')
```

will do so if the DEBUG has been set for this event. KEPRNT should be called at the start of each module. The proposed action for PRINT key values is:

0	no output
1	max. 6 lines / module (typically entry and exit)
2	max. 50 lines / module (one page)
3	full debug print-out

The current list of print options is as follows:

key	module	value	action
DUMP	READIN	1	print no. of records, unknown label *
		2	print catalogue of label read,trigger bit, event tag *
		4	print all banks which are read/written *
FRAM	frame	0-2	general CHANT framework printout level
INOU	frame	0-2	CHANT I/O printout level
PRER	frame	0-6	event-by-event error printout from ERREPO
CHIO	chIO	0-5	chIO package printout level
EDIR	xxxxxx	2	xxxxx
SKIP	IMODUL	1	print catalog, scaler bank of skipped events
RUNS	RUNSTA RUNEND	2	print run start and end summary
TIME	PROCES	1	print CPU consumption for each event

		2	create n-tuple with event-by-event CPU consumption
		4	print CPU consumption for each module for each event
		7	- 1+2+4
DZVE	ALLMOD	1	ZEBRA storage integrity check once per event
		2	same check once per module per event
TERM	BCHARS	80	set display line width to 80 (default 132)
		3	for 'big character title' (see TITLE)

OPTION has the same format as PRINT, and allows options and associated values to be read, and tested by KAFLAG (like KAPRNT). Implemented so far:

OPTION 'NODB' 1 switch off database access

OPTION 'TRIG' n selects events according to their trigger bit pattern. Valid trigger bits for global events (LABEL=111) are

Bit	meaning
1	pilot target
2	main emulsion
3	auxiliary emulsion
4	random clock
5	strobe (beam muons without OPTO readout)
6	testbeam
7	pulser
8	alignment
9	calorimeter minimum bias
10	calorimeter charged current
11	calorimeter two track
12	calorimeter quasi elastics
13	spectrometer minimum bias
14	spectrometer charged current
15	spectrometer two track
16	spectrometer heavy lepton

(e.g. OPTION 'TRIG' 6 selects trigger bits 2 and 3 - emulsion trigger) For the events with LABEL=134 (local OPTO data) the trigger word has a different meaning. Now it has to be interpreted as a whole integer number, not bit by bit, and the numbers refer to the following event types:

1	cosmic
2	CCD threshold event (pedestal)

3	high intensity fiducial
4	low intensity fiducial
5	beta ray
6	low intensity fiducial with variable MCP gate

OPTION 'XFER' -iI handles so-called overlaid OPTO events. Special alignment runs have been taken during which several events are overlaid in the opto-electronics readout of the fibre trackers (TT and DT). This option is enabled with:

```
*DO OPTION -iI
:TRIG 128 :XFER 1
```

It should be noted that this option requires the trigger module. This module reads the actual trigger settings from the input data (LABEL 209). Therefore, the following data card entries are mandatory:

```
*DO MODU -iI
:FUJI :DEFA
:FTRA :DEFA
:TRIG :DEFA

*DO LABEL -iI
201 1 202 1 203 1 204 0 206 0 207 0 208 0 209 1 305 0
130 0 131 0 132 0 133 0 134 0 135 0 121 1
111 1
```

A print option is available (e.g. `:XFER 5`, see above).

OPTION 'ERRO' n allows to suppress job termination or "QNEXT recovery" requested by call to ERREPO with severity levels "T" and "F". $n = 1$ to disable "call termin", $n = 2$ to disable "call qnext", $n = 3$ to disable both.

OPTION 'DBPT' n provide bit pattern (integer n) when requesting event from ChorusDB Objectivity/TM database.

5.3 Steering cards supported by reconstruction modules

SPECT - a muon spectrometer reconstruction module.

The card **SPEC** allows switching off the parts of the spectrometer unpacking and reconstruction algorithms. It should be used by experts for debug purposes only.

```
*DO SPEC -iI
:UNTT n :UNAN n :UNTU n :BLFI n :SAMT n
```

Where “n” can be either “0” (off) or “1” (on). If the card is missing, all SPEC submodules are “ON”.

The following **PRINT** flags are used in SPECT:

key	module	value	action
SAMT	SAMTRA	n	print output of SAMTRA for several levels : 0 = nothing until 9 = debugging
SPEC	DATSPC	n	debug muon finding and fitting
SBTC	SPCBTC	1-2	debug best-momentum fit routine
HYVE	HYVESP	n	debug broken line fit in muon spectrometer
CMMR	CAMMOR	n	debug muon momentum from range calculation
UNBR	UNBREM		unpack end system scintillator data
UNDR	UNDRIF		unpack drift chamber data
DECC	DEDRIF	1	print entry and exit messages
		3	dump input and output banks
DEDR	DEDRIF	n	equivalent to DECC
DSPC	DATSPC	n	debug spectrometer streamer tube tracking

The following **OPTIONs** are used in SPECT:

:SPMG n specifies the maximum number of magnets, allowed to be skipped by the “skip gap” algorithm. :SPMG -1 disables “skip gap”.

:SPMU n specifies the number of muons, the “skip gap” algorithm is trying to find. :SPMU -1 disables “skip gap”.

If SPMG and SPMU are missing or both are set to zero, the defaults, 5 and 1, will be used correspondingly.

HXTR - The hexagonal magnet track reconstruction module (HEXTRA).

The card **HXTR** allows switching on/off the usage of Honeycomb track information.

```
*DO HXTR -iI
:MODE n
```

Where “n” can be either “-1” (off), “+1” (on) or “0”. If the card is missing, the default value of 0 is assumed. If the operation mode is 0, it is redefined on the basis of the HEPDB contents. If the geometry and calibration parameters of the Honeycomb chambers are found for the current run, the operation mode is set to +1, otherwise to -1.

The following **PRINT** flags are used in the HEXTRA:

key	module	value	action
HXTR	HEXTRA	n	print output of HEXTRA for several levels : 0 = nothing until 5 = full debugging

???????? - xxxx yyyy reconstruction module.

The following print flags are used in ????????:

key	module	value	action
FTSP	UNFTSP	n	1 = entry/exit message ; 2 = 1 line per chain
DTSP	UNDTSP	n	1 = entry/exit message ; 2 = 1 line per chain
BRAY	UNBRAY	n	debug beta-ray handling
SGTT	SGTATR	n	debug fiber tracker tracking
TTEX	SGTATR	n	debug fiber tracker extrapolation
SGDT	SGDTTR	n	debug diamond tracker tracking
VTEX	VERTEX	n	debug fiber tracker vertex finding
ECAL	CALOR	n	debug calorimeter energy calculation
UNSC	UNCALO		unpack calorimeter scintillator data
QERC	QEVERT	n	debug Q.E. vertex recovery
QECA	QEVERT	n	debug calorimeter handling of Q.E. vyx recovery
CLUN	CLUNET	n	debug calorimeter clustering
TCAL	CATRAS	n	debug calorimeter streamer tube tracking
EMUL	EMSCAN	n	debug output of scanning predictions
DTNT	SGDTTR	1	creates histogram file
CCSL	CCSELE	n	debug CC/NC selection
OVER	OVERLAY	n	debug CCD data overlay handling
TTMU	TTMURE	n	debug fiber tracker muon recovery
TTHC	TTHCRE	n	debug fiber tracker HC recovery
CLUS	UNFITR	n	CCD track element search
TROU	TRIGGER	n	produce output from trigger banks
UNTH	UNTRIG		unpack trigger banks
STRK	VERTNW	n	debug vertex refit, track recovery

SLMC	SELMCEV	**T**	debug selection of MC channel
UNPA	UNPACK	1	print entry and exit messages
		3	dump input and output banks
UNTC	UNTUCD		unpack calorimeter streamer tube data
UNFT	UNFITR		unpack opto data
UNDI	UNDISN		unpack opto fiducial events
UNBM	UNBEAM		unpack beam data

Chapter 6

Writing a new CHANT module

This chapter describes how to integrate a new module in CHANT.

When a new module is being prepared, one should observe the following:

- The module should be as independent as possible. This means, in particular:
  - communication with other modules are allowed only via ZEBRA banks and return code of the main module routine;
  - communication with CHANT frame is allowed also by the common blocks /XQSTOR/, /RUNCOM/ and /RAWHLC/;
  - navigation through the event bank tree is performed via CHANT utility functions (see section XXX);
  - pointers to Chorus data-base banks are obtained via MCDUSE routine;
  - all needed pointers are kept (and MZLINK-ed) in local common blocks;
  - the module defines, if needed, additional data cards (via ZEBRA TZ package) for control and initialization purposes.
- A "data-line" for the module's action has to be included in CHANT frame MODU card (see below).
- The module should create its own bank tree only under MODU bank, which is a first daughter of EVNT top-level bank. Temporary banks can be created anywhere and must be dropped before exit.
- Invocation of the module has to be ensured by a call to it in *main/src/allmod.F*. On exit, it should return an error code.

There is a template module in CHANT package, written just to facilitate integration of new modules. The code is in directory *chant/tpmo/*. Fortran *.F files are in *chant/tpmo/src*, #include files - in *chant/tpmo/comm* and a run script with data cards - in *chant/tpmo/doc*. This is a real example, which compiles and runs. The developer is strongly advised to examine the *tpmo* routines . They could well serve as a "blank form" to be filled in with some reconstruction code. The calling tree of the module is presented below along with some comments about each routine. More comments can be found in the code itself.

```
TPMAIN      ---  Template module main routine.
|                The name of the module is 'TPMO' and it is used to compose
|                bank names and as a string identifier in steering cards.
+- FRACTN   ---  Integer function. Returns action code.
|                According to it, the module decides what to do further.
+- TPACTN   ---  looks for and get data from module's own steering card.
|                The card identifier is chosen to be 'TPMA'
|                (TemPlate Module Action).
|                One can define here his/her own flags to control
|                the behaviour of the module.
+- TPLNK    ---  initializes module's link areas
+- TPINIT   ---  Initialization routine. MZFORM for the banks is here.
|                It looks also for and get data from another steering
|                card with some constants. The card identifier is
|                chosen to be 'TPMC' (TemPlate Module Constants).
+- TPMODB   ---  invokes MCDUSE for HEPDB banks needed by the module.
|       MCDUSE   (look at the documentation for MCDUSE in this manual).
|
+- TPCALC    --- The routine that does the main job for the module
|       CUTBNK .
|       ERREPO  .
|       EVFANO   . ---  utility routines, provided by CHANTframe
|       IGETBT .
|
|       TPSUB1 --- (Empty) subroutines of TPCALC
|       TPSUB2
|
+- ERREPO.
+- EVCLOS .
+- FRACTN  . ---  utility routines, provided by CHANTframe
+- KAPRNT.
```

Following steering cards are needed for the module:

```
*DO MODU -iI      #. This card defines actions of all modules.
...

:TPMO :DEFA       #. This defines action for template module.
                  #. It can be omitted if DEFA is desired behaviour.
...
```

```
*DO TPMA -iI      #. template module steering card
:TPS1 1           #. activate submodule 1
:TPS2 1           #. activate submodule 2

*DO TPMC -iI      #. Initialization constants for the template module.
                  #. (empty: no constants).
```

For compilation and linking, one should examine section **3.2 Installation and compilation for developer/librarian**.

Chapter 7

Interactive event display

The interactive display has been implemented as a seperate module (`tvshow`) which is activated with the following option in the data card:

```
*DO MODU -iI     #. ON OFF DEFA FORC
:SHOW :DEFA
```

or using **chant-show-event** shell script.

User interface of the display is based on KUIP package. For more details about KUIP and TVSHOW - use HELP command.

The event display can show global events as well as burst events. At run start, it will look for file `higz_windows.dat` in the current directory. If this file doesn't exist, it will be created. The first line of this file is used to define the window size and its location on the terminal screen. The window size can be redefined with the standard mouse action. The graphics will be then lost. It can be recovered with the `PLOT` command (see below). All displayed events are also written to file: `f.%run.%event.eps`. Redisplaying the same event overwrites the current output file.

Running chant event display will show the first event and prompt:

```
TVSHOW>
```

Possible commands:

`STOP`	Terminate chant
`NEXT`	Go to the next event
`PLOT`	Plot same event (after window action)
`SHOW`	List current parameter values
`ZOOM`	Zoom in/out of detector view
`VIEW`	Define projection angles
`FULL`	Full event display (default)
`TRACKS`	Define track types to be shown in target region
`PRINF`	Set printout level for different kinds of information
`GRINF`	Set printout level for different kinds of information in the graphic window
`FIND`	Goto run / event and display
`<RETURN>`	Next event

The following commands have additional arguments (`[]` are optional):

`ZOOM XMIN [XMAX [YMIN [YMAX]]]`
Outer detector coordinates in cm. Without arguments, interactive (mouse-driven) zoom is invoked. It can be done independently on two projections.

`VIEW PROJ(1) [PROJ(2)]`
Projection angle of top(1) and bottom(2) view in radians.

`FIND [IRUN] IEVENT`
Run `IRUN` and event `IEVENT`

The address in the `TZIN` command can be an integer or a character string starting with ':'. The bank will be overwritten starting at the location next to the data word matching the address `WORD`. The new value can be a float, an integer or a character string starting with ':'. No range checking has been implemented!

Running the chant event display will create the file `tvshow.kumac` in the current directory. It contains the commands entered by the user. Any file containing commands can be processed by entering `EXEC file.kumac`. It is not possible to have nested `EXEC` commands.

The routine `TVUSER` is called at the end of the event display. This allows the user to display extra graphics output.

Examples

The target tracker display is obtained with:
`ZOOM -20 +50 -100 +100`

The hits of the inclined honeycomb chambers and spectrometer drift chambers are shown with:
`VIEW -0.52 +0.52`

The hits of the inclined planes of the fibertrackers are shown with:
`VIEW -0.14 +0.14`

`TZIN PRINT :HXTR 5` maximises the debug output of the `HEXTRA` module and `TZIN LABEL 203 0` switches off the processing of burst events. No arguments to the `TZIN` command starts `TZIN` interactively. This is useful to create new `TZ` entries. It should be stopped by entering `*FIN`.

CCD image display

The interactive CCD image display can be started by entering `GOTO FUJI`. It changes the prompt to:

```
CCD>
```

Possible commands are now:

`QUIT`	Terminate chant
`HELP`	List possible commands
`BACK`	Return to event display
`CCD`	Display raw CCD image
`OPEN`	Start display CCD image
`<RETURN>`	Return to `TVSHOW` event display

The following commands have additional arguments (`[]` are optional):

`CCD  CHAIN`
  Chain number `[1-58]`.

`OPEN  [file] CHAIN`
  File name and chain number `[1-58]`.

After the OPEN command, the user can add additional graphics material.

```
CCD> ADD
```

with the following options:

FUJI Add output of FUJI module.
FTRA Add output of FTRA module.

The FTRA option has an additional (optional) argument ITRAK (track number), which allows to display a given track. The default is 0 which displays all tracks reconstructed by FTRA.

After issuing the OPEN command, the user should close the CCD display with the command CLOSE.

Appendix A

Coordinate system

For the description of the detector geometry a common coordinate system is used. It is a right-handed Cartesian system with the following axis orientations:

GEANT	CHANT	
z-axis	x-axis	horizontal projection of neutrino beam axis
x-axis	y-axis	horizontal from Gex towards Bellegard
y-axis	z-axis	vertical upwards

The difference in respect to the GEANT conventions is there for historical reasons. Nevertheless there shouldn't be any transformation problem between GEANT and CHANT because only a cyclic change of the coordinates has to be performed. The origin of the coordinate system was chosen to be unique for both the systems and is defined to be a symmetry centre between the 4 emulsion stacks which sits on the theoretical beam axis. Most of the surveys are done with respect to the reference point 'centre of BEBC hall' therefore the following transformation with respect to this position has to be applied:

$$X_{CHANT} = X_{BEBC} + 5.9266m$$

$$Y_{CHANT} = Y_{BEBC}$$

$$Z_{CHANT} = Z_{BEBC} + 0.2532m$$

The difference in z can be calculated from the beam slope 0.04273 (tangent of beam angle) and the difference in x because the 'canter of BEBC hall' is as well located on the theoretical beam axis.

Appendix B

Practical examples

B.1 Steering cards for official MAXI-DST production (as of 1.01.2001)

```
*PRINT #.print this D/C on stdOUT#
*DO FILI -AW     #. input file(s)
:/local/chprod/input/1996.21-Jan-2001/input.dst1-cs21.6073.fz
*DO FILO -AW     #. output file(s)
:MAXI :/local/chprod/output/1996.21-Jan-2001/dst1.6073.fz


*DO MODU -iI     #. ON OFF DEFA FORC
:ALL  :DEFA
:TPMO :OFF
:SHOW :OFF
:KREN :OFF
:EMFB :OFF
:HXTR :OFF
*DO TITLE -iI
:chant

*DO DEBUG -iI
1 -50000

*DO LABEL -iI
130 0 131 0 132 0 133 0 134 0 135 0
```

```
*DO OPTION -iI
:TRIG 6

*DO PRINT -iI
:FRAM 0 :INOU 0 :KEYS 0 :DUMP 0 :SKIP 0 :RUNS 0 :SLMC 0
:TERM 132 :EDIR 0 :CHIO 1 :DZVE 0 :FUJI 0 :UNDI 0 :MSPC 0 :TIME 1
:UNTT 0 :UNDR 0 :DEDR 0 :DSPC 0 :SPEC 0 :SAMT 0 :UNAN 0 :FTRA 0
:DTSP 1   #.This will produce only a message that DT is not yet included.

*DO PRED -iI -AC            #. PRED steering card
:ASCI 92 :/local/chprod/output/1996.21-Jan-2001/predictions.6073.ascii #. Ascii i/o unit

*DO COND -iI
:CERR 99999  :TIME 50.

*DO DEVENT -iI
1 -9999999

*DO DRUN -iI
1 -9999999

*DO CEVENT -iI
1 -9999999

*DO CRUN -iI
1 -999999


*FINISH #.end of file#
```

B.2 An example of the shell script to run CHANT

```
In this example everything after the ‘‘<<***EoF***’’
goes to ‘‘stdin’’ of CHANT and contains its steering cards.

#!/bin/sh
```

```
./bin/chant>mychant.out <<***EoF***

*PRINT #.print this D/C on stdOUT#

*DO FILI -AW     #. input file(s)
"chstagein Y20422.37"

*DO FILO -AW     #. output file(s)
:MAXI :/afs/cern.ch/chorus/scratch/week/xxxx/my_maxi.fz
:USER :/afs/cern.ch/chorus/scratch/week/xxxx/my_user.fz

*DO USTR -iI        #. USER stream control
:RAWH 0 :GLOB 0 :MCEV 0 :UNPK 0 :FTRH 0

*DO MODU -iI     #. ON OFF DEFA FORC
:ALL  :DEFA
:MIDI :OFF
:HONC :OFF
:SPEC :FORC
:KREN :OFF
:SHOW :OFF
:TPMO :OFF
:EMFB :OFF
*. :HXTR :OFF

*DO DEBUG -iI
1 -20000

*DO TITLE -iI
:chant

*DO LABEL -iI
130 0 131 0 132 0 133 0 134 0 135 0
*. 201 1 202 1 203 1 204 1 206 1 207 0 208 0 209 0 305 0

*DO OPTION -iI
:NODB 0 :TRIG 0


*DO PRINT -iI
:FRAM 1 :INOU 0 :KEYS 0 :DUMP 0 :SKIP 0 :RUNS 0 :SLMC 0 :UNSC 0
```

```
:TERM 132 :EDIR 0 :CHIO 1 :DZVE 0 :UNPA 0 :FTRA 0 :ECAL 0 :FUJI 0
:UNTT 0 :UNDR 0 :DEDR 0 :DSPC 0 :SPEC 0 :SAMT 0 :UNAN 0 :GRAF 0
:MSTV 0 :MSPC 0 :TIME 1

*DO COND -iI
:CERR 99999  :TIME 50.

*DO DEVENT -iI
1 -999999

*DO DRUN -iI
1 -999999

*DO CEVENT -iI
1 -99999

*DO CRUN -iI
1 -99999

*FINISH #.end of file#
```

B.3 An example of UNIX session where a new user builds his own CHANT executable and runs it for the first time

```
mkdir example
cd example
mkdir obj

export CHORUS_ROOT=$CHORUS/dev
export ROOT_DIR=./

cp $CHORUS_ROOT/src/chant/user_Makefile Makefile
cp -r $CHORUS_ROOT/src/chant/user ./
cp $CHORUS_ROOT/src/chant/doc/modusr.F_example modusr.F
cp $CHORUS_ROOT/src/chant/doc/cpp_modusr.cpp_example cpp_modusr.cpp
cp $CHORUS_ROOT/src/chant/doc/run.chant ./
```

```
gmake chant

chmod a+x ./run.chant
./run.chant&
```

When the job will be finished (the processing itself takes a couple of minutes, but if the tape is not staged, it may take more) you can look on the *chant.out* file where some information extracted from spectrometer banks is printed, as well as a couple of histograms. File *modusr.F* (Fortran) and *cpp_modusr.cpp* (C++) contain examples of user code to access data from ZEBRA banks. You can now modify them according to your needs.

Another example of user's code, illustrating access to CCD microcluster information, can be found on

```
$CHORUS_ROOT/src/chant/doc/modusr.F_get_muclusters
```

Index