NOTE 233 – Guide to Measures in C++

Wim Brouw

20 May 2000

A pdf version of this note is available.

Contents

1 Introduction **2**

2 Background **2**

3 Conversion **4**

4 Efficiency **5**

4.1 Measure or MeasValue 5

4.2 Re-use the effort of frame 6

4.3 Re-use of conversion objects 7

4.4 Specialized conversion engines 7

4.4.1 EarthMagneticMachine 7

4.4.2 VelocityMachine 8

4.4.3 UVWMachine 10

4.5 Specifying execution details 11

5 Interface **12**

5.1 Getting values from a Measure 12

5.2 Getting values from a MeasValue 13

5.3 Obtaining information from the MeasFrame 13

A Appendix **15**

A.1 Background literature 15

A.2 Details for individual Measures 15

A.2.1 MEpoch 16

A.2.2 MPosition 16

A.2.3 MDirection 17
A.2.4 MBaseline 19
A.2.5 Muvw 19
A.2.6 MEarthMagnetic 20
A.2.7 MFrequency 21
A.2.8 MRadialVelocity 22
A.2.9 MDoppler 23

1 Introduction

This guide gives a short overview of the background and use of the *Measures* module in C++. Detailed information can be found in the

Measures module description, and in the individual header files of the classes in this module.

Section 2 explains what a *Measure* consists of, and why; section 3 discusses conversion; section 4 some notes on efficient use and section 5 gives the common interface of *Measures* and their values. Appendices give some details per *Measure*.

2 Background

Currently the following measures exist:

$Epoch$ a high precision epoch, a moment in time

$Position$ a 3-dimensional position vector, in general indicating a position w.r.t. Earth

$Direction$ a direction in space (i.e. a position of unit length)

$Frequency$ the frequency of an electro-magnetic wave

$Doppler$ the redshift of an object

$RadialVelocity$ velocity along a direction

$EarthMagneticField$ the earth magnetic field vector

$Baseline$ a direction with a length

uvw a u, v, w coordinate: a baseline projected on some plane

A specific *Measure* class is indicated with an M, e.g. a direction measure is an $MDirection$.

A *Measure* consists of a VALUE and a REFERENCE. The VALUE of a *Measure* is a vector of double floating point numbers, in some internal format, independent of the units used to create them. The VALUE is called a *MeasValue*, and specific ones are indicated with the letters MV, e.g. $MVDirection$ [1]. Appendix A gives the various internal values.

[1]Note the unfortunate circumstance that the classes *MVTime* and *MVAngle*, which are not *MeasValues*, but only formatting classes for times and angles, were named as such.

A VALUE can be constructed in many different ways. An *MVDirection* can, e.g. be constructed from 2 angles or from 3 direction cosines; an *MVFrequency* from a wavelength or the wave energy.

Some *Measures* can often also be obtained from a catalog, especially a list of observatories, a source list and a spectral line list. They can be obtained as:

```
MDirection source;
MFrequency line;
MPosition obs;
if (MeasTable::Observatory(obs, "ATCA")) {};
if (MeasTable::Line(line, "HI")) {};
if (MeasTable::Source(source, "0008-421")) {};
```

The REFERENCE consists of up to 3 *fields*:

- a *reference code*, describing how the VALUE should be interpreted. This field is mandatory, and has an enumerated value (see Appendix A for the possible codes). Examples are:
  - `MDirection::J2000`
  - `MDirection::AZEL`
  - `MEpoch::TAI`
  - `MEarthMagnetic::DEFAULT`

  In some cases the code has all the available information to be able to interpret the VALUE (like the `MDirection::J2000`). In other cases the information is not sufficient. An example is the `MDirection::AZEL`. If you have measured some *AZEL* values, and are only interested in say a pointing model for the telescope, the `MDirection::AZEL` is sufficient information. However, if you want to convert the measured *AZEL* into a *J2000* right ascension and declination, it is necessary to know where your telescope is, and what the epoch of the observation is. This can be specified in

- a *frame*. The *frame* is a *MeasFrame* class object, and can contain *Measures* describing the epoch, the position on Earth, the direction in which you are observing (for Doppler calculations), the rest frequency of a spectral line and a table describing the detailed orbit of a non-standard solar system object (like a comet). The *frame* can be filled with the *MeasFrame* constructor, or with *MeasFrame* `set()` methods.

- an optional *offset*. In some cases it can be advantageous to specify an offset. An example could be 0h on a specific date, with the actual *Epoch* only specifying the time since that point in time. Note that offsets are not very useful for *Directions*, since directions are always vectors of unit lengths. However, special `offset` methods exist for directions.

A REFERENCE is created as a specialized *MeasRef* object, using the `Measure::Ref` alias (e.g. `MDirection::Ref`). A full reference could be:

```
// A time in MJD
MEpoch epoch(Quantity("50500.5d"), MEpoch::UTC);
MPosition obs;
MeasTable::Observatory(obs, "WSRT");
MeasFrame frame(obs, epoch);
MDirection::Ref ref(MDirection::VENUS, frame);
```

3 Conversion

Measures can be converted from one REFERENCE into another REFERENCE. E.g. from an `MDirection::J2000` direction coordinate into an `MDirection::AZEL` coordinate; or from an `MEpoch::UTC` into an `MEpoch::TAI`. Conversion objects are `MeasConvert` objects, using the `Measure::Convert` alias (e.g. `MDirection::Convert`. The conversion object needs at least a *from* REFERENCE and a *to* REFERENCE.

Example:

```
MDirection::Convert(MDirection::Ref(MDirection::VENUS,
                                    frame),
                    MDirection::Ref(MDirection::J2000));
```

Note that if a *frame* is necessary, it suffices to give it with either one of the two REFERENCES. Unless, of course, they are different for the two REFERENCES, like converting the `AZEL` at one telescope to that at another. The latter case (i.e. two different reference frames for the input and output values), is handled by always converting first the input value to the default for the class (e.g. `J2000` for directions), and then convert this value to the appropriate output type and frame.

The *from* REFERENCE can also be specified as a complete *Measure*. In that case that VALUE of the *Measure* will act as the *default* VALUE to be converted.

Constructing a conversion object will set up a series of conversions that have to be done to get from the *from* to the *to* REFERENCE. This state-machine like approach is to be able to use less than the required odd 400 conversion routines for say 20 different allowable reference codes.

The conversion object is executed with the () operator. The actual conversion done depends on the argument of the operator:

- *none*: use the default value of the Measure specified in the constructor (or using a `set()` method).
- *MeasValue*: convert it to a *Measure* using the *from* reference of the object
- *Measure*: re-initialize the conversion object, and do the conversion with the new default value (since the REFERENCE has changed!), taking the full *Measure* into account (including, of course, a possible reference frame).

If the conversion needs information, it will cache anything it calculates, either in the conversion object (like calculated *Nutation*), or in the *frame* (like e.g. the sidereal time of a frame MEpoch specified in UTC). This information will be re-used if possible and feasible in subsequent conversions with the same conversion object or using the same frame.

It is also worth noting that *frames* are handled internally by *reference* rather than by *value* (e.g. when copied) (the same is true for `Measure::Ref` objects). One consequence is that a conversion knows about any change you make to a frame, and will use it. So, if by a `frame.set(MEpoch)` the time of a frame is changed, a subsequent conversion which had that frame given as the frame to be used, will automatically use the new time.

4 Efficiency

The efficiency of the use of the *Measures* and related classes can vary greatly. By using the appropriate interface, by fine-tuning with the aid of `aipsrc` variables, and by making sure the caching system is used optimally, there can be a large reduction in resource use.

4.1 Measure or MeasValue

It is not always necessary to use a full-fledged *Measure*. It often suffices to use a *MeasValue* (or maybe even a *Quantity* or just a simple *Double*.

As an example, consider a frequency container object:

- if you have some box that produces a value that is always in GHz, and that has to be passed-on as-is without any further information, a *Double* is sufficient
- if the value you have has to be converted to some other frequency unit (like MHz or so) before being used, use a *Quantity*. The only knowledge necessary is the units
- if you would like the value be represented in other ways to represent an electro-magnetic wave (like wavelength, time, energy, impulse, angle/time, time), use an *MVFrequency*, which knows that the value has the properties of an electro-magnetic wave (in addition to the units)
- if you would like to know what the frequency really represent (a spectral rest frequency; a frequency as observed in a telescope; a frequency as would be observed from the local standard of rest; ...), and if you are going to use that information in one way or another (display, conversion, archive, ...) use an *MFrequency* as value container

As a rule of thumb the above could be summarised as:

- designing a user interface: be flexible, use a *Measure*
- designing a hardware interface: use a *Double* or *Int*
- in internal classes, often use *MeasValue*, but will depend on how close you are to user interface or another internal interface

Similar arguments can be used for the other *Measures*.

4.2 Re-use the effort of frame

From the point of view of the programmer, a *frame* is just a container of *Measures* to indicate when, where, in what direction and at what frequency a certain *Measure* was made or referred to. However, in actual fact it is also an engine and cache for a lot of calculations. Imagine that you want to convert from right-ascension and declination to hour-angle and declination, and that you have provided an epoch in UTC in the *frame*. The actual conversion object will request (probably among other things), the sidereal time from the frame. The first request will set up a conversion object within the frame (from UTC to $LAST$) and cache it for later use. After that it will use this conversion object to obtain the sidereal time, and cache the result (for maybe a subsequent call). The conversion needs nutation, polar motion and a few other calculations. Again, all of these are cached for subsequent use in other calls to the frame for information.

Efficient use of *Measures* is only possible if the lifetime of a *frame* is as long as possible. Which suggests that in many cases a frame should be created at the highest level possible (maybe even globally). Re-use of a *frame*, e.g. for a different time, is made possible by the `set()` methods, e.g. *set*(*MEpoch*), which will try to minimize the re-calculations necessary. The fact that frames are always used internally by *reference* (see earlier), which also means that any change made to a frame will automatically be used by any subsequent conversion which knows about this frame, makes in principle for an efficient machinery for many different purposes. However, it could also easily lead to misunderstandings. If you plan to do any special conversions, the detailed information provided in the various `Measure` classes should be perused.

4.3 Re-use of conversion objects

Similar to the frame discussed in the previous section, a conversion object (e.g. `MDirection::Convert`) is a repository of the necessary state-machine to make the conversion possible, and any intermediate calculation results. To make efficient use of this information, the same conversion object should be used if more than one conversion of the same type has to be done.

4.4 Specialized conversion engines

Although the official `MeasConvert` objects are very versatile, using them can be quite a job. For that reason a set of specialized "Conversion engines" have been put together for easy use. The three engines available at the moment are described in the following paragraphs. They have all a basic format:

- a constructor is used to create an engine object. The object knows (either through constructor, or through the use of `set()` methods) about the conversions to be done
- the *()* operator is used to convert a value the appropriate way.

4.4.1 EarthMagneticMachine

The *EarthMagneticMachine* calculates the Earth' magnetic field in a certain direction at a certain height above the Earth' surface.

The machine object's constructor needs in principle:

- a direction *reference code* to be able to interpret input direction values
- height above the Earth' surface
- position (of observation point) on Earth
- epoch of observation (the $IGRF$ model used is time dependent)

The () operator will produce the line-of-sight component of the magnetic field (see the header files for details). Other methods exist to get the complete magnetic field; the longitude of the point specified and the position on Earth of the point. The following example calculates the magnetic field at $200km$ height at the Compact Array:

```
// Define a time/position frame
MEpoch epo(MVEpoch(MVTime(98,5,16,0.5).day()));
MPosition pos;
MeasTable::Observatory(pos, "ATCA");
MeasFrame frame(epo, pos);
// Note that the time in the frame can be changed later
// Set up a machine
EarthMagneticMachine exec(MDirection::B1950,
                          Quantity(200, "km"), frame);
// Given a current observational direction
MDirection indir(Quantity(3.25745692, "rad"),
                 Quantity(0.040643336,"rad"),
                 MDirection::Ref(MDirection::B1950));
// The field in this direction is calculated
exec.calculate(indir.getValue());
// Show some data
cout << "Parallel field: " << exec.getLOSField() <<
        " nT" << endl;
cout << "Sub-ionosphere long: " << exec.getLong("deg") <<
        endl;
```

4.4.2 VelocityMachine

The velocity machine converts between frequencies and velocities (or *vice versa*). This machine has been developed to aid in the, especially for the beginning user of *Measures*, intricate way *RadialVelocity*, *Doppler* and *Frequency* are connected. The machine converts between *Doppler* and $Frequency$ values.

The constructor of the machine needs:

- a frequency reference (e.g. `MFrequency::LSR`), including a possible offset and frame
- preferred frequency units (e.g. *cm* or *GHz*)
- velocity reference (e.g. `MDoppler::OPTICAL`), including a possible offset
- preferred velocity units (e.g. AU/a)
- the rest frequency used for the conversion, given as an `MVFrequency`

The () operator has an `MVFrequency`, an `MVDoppler` or a `Quantity` as argument. Depending on if it is a velocity or a frequency, the argument is converted to the other representation. `makeFrequency` and `makeVelocity` exist to create a vector of velocities or frequencies from a vector of Doubles.

An example:

```
// Define a time/position frame
MEpoch epo(MVEpoch(MVTime(98,5,16,0.5).day()));
MPosition pos;
MeasTable::Observatory(pos, "ATCA");
MeasFrame frame(epo, pos);
//
// Note that the time in the frame can be changed later
// Specify the frequency reference
MFrequency::Ref fr(MFrequency::LSR);
//
// Specify the velocity reference
MDoppler::Ref vr(MDoppler::OPT);
//
// Specify the default units
Unit fu("eV");
Unit vu("AU/a");
//
// Get the rest frequency
MVFrequency rfrq(QC::HI);
//
// Set up a machine (no conversion of reference frame)
VelocityMachine exec(fr, fu, rfrq, vr, vu, frame);
//
// or as (with conversion of reference frame it
// could have been)
// VelocityMachine exec(fr, fu, rfrq, MFrequency::TOPO,
//                      vr, vu, frame);
// Given a current observational frequency of
//    5.87432837e-06 eV
// its velocity will be (in AU/yr)
cout << "Velocity: " <<
        exec.makeVelocity(5.87432837e-06) << endl;
//
// Introducing an offset
MFrequency foff(MVFrequency(Quantity(5.87432837e-06,
                "eV")), MFrequency::LSR);
//
// and setting it in the reference, and regenerating
// machine:
fr.set(foff);
exec.set(fr);
//
// the following will give the same result:
cout << "Velocity: " << exec.makeVelocity(0.0)
     << endl;
```

4.4.3 UVWMachine

The *UVWMachine* can convert *UVW*-coordinates between coordinate systems. In addition it can provide the phase rotation necessary on the data to have a new fringe-stopping center. A simple conversion of *UVW* coordinates will be executed if only the coordinate reference frame is changed (e.g. from a *J*2000 to a *Galactic* or an *AzEl* coordinate system). If also the actual position on the sky is changed, the phase rotation necessary on the data is provided as well. Fringe stopping centers can be centered on other bodies as well (e.g. a planet). Read the caveats in the detailed help file.

The constructor of the machine needs the following input:

- an input `MDirection` specifying the original fringe stopping center's position and reference frame
- an output `reference code` to indicate the output reference frame of the *UVW* coordinates; or an output `MDirection` indicating both the new fringe stopping center and its reference position.

The output of the machine can be one or all of the following:

- a rotation matrix that can be used to transpose the UVW-coordinates
- a vector that can be used to produce the necessary phase rotation
- actual conversion of a set of input UVW points
- actual vector of phase rotations for a set of UVW points

Example:

```
// Given a current phase stopping Center
MDirection indir(Quantity(3.25745692, "rad"),
                 Quantity(0.040643336,"rad"),
                 MDirection::Ref(MDirection::B1950));
// Conversion to J2000 is set by:
UVWMachine uvm(MDirection::Ref(MDirection::J2000), indir);
// The rotation matrix to go to new UVW is obtained by:
RotMatrix rm(uvm.rotationUVM());
// If an UVW specified:
MVPosition uvw(-739.048461, -1939.10604, 1168.62562);
// This can be converted by e.g.:
uvw *= rm;
// Or, alternatively, by e.g.:
uvm.convertUVW(uvw);
```

4.5 Specifying execution details

The precision of a *Measure* conversion can be influenced by the use of *aipsrc* variables. Changing the precision will influence the efficiency of the *Measures* conversion.

Parameters that can be used are, a.o., the length of periods over which computations can be re-used; if it is necessary to use $IERS$ tables for precise calculation of time and polar motion, or that the simple model suffices.

A few examples:

Name	Description	Default
measures.nutation.d_interval	interval in days over which linear interpolation of nutation calculation is appropriate	0.04d
measures.nutation.b_useiers	use the IERS Earth orientation parameters tables to calculate nutation	false
measures.nutation.b_usejpl	use the JPL DE database (use measures.jpl.ephemeris to specify which one) to calculate nutation	false
measures.measiers.b_notable	do not use the IERSeop97 or IERSpredict tables at all for calculations	false

5 Interface

The interface to the different *Measures* have a large set of identical methods. Only when values are clearly related to a specific *Measure* (like e.g. a latitude) does the method to obtain not exist for e.g. a frequency. Constructors of *Measures* have often specializations (like one with a longitude and latitude for a direction), but they all have also standard ones. The standard ones are described in the
Measures.h

5.1 Getting values from a Measure

The various value that are contained in the *Measure* object can be obtained by the following calls:

getValue() will obtain the *MeasValue* (i.e. the *MVName* object) from the *Measure*

getData() will obtain a pointer to the *MeasValue*

getRef() will obtain the `Measure::Ref` reference object

getRefPtr() will obtain a pointer to the `Measure::Ref` object

get() defined for some to get the data in special form (with specifiable units

getAngle() defined for some to get the data in angle (with specifiable units)

General calls are available to obtain the information contained in the `Measure::Ref` object:

$getType()$ will obtain the reference code from the `Measure::Ref` object

$getFrame()$ will obtain the frame from the `Measure::Ref` object

$offset()$ will return the pointer to the offset (or 0 if no offset specified)

`set()` methods are available to fill in the fields in the `Measure::Ref` object.

5.2 Getting values from a MeasValue

Each `MeasValue` has methods to obtain the internal data in a standard way. Some have additional ones available when appropriate (like *getLong()*):

$getVector()$ will always return the internal value as a `Vector<Double>`

$getValue$ will obtain the internal value as a *Double* or as a `Vector<Double>`, depending on the dimension of the internal value. The units of the value returned will be the internally used ones (e.g. m for a position, s for a time.

$get()$ will obtain the internal value as a *Quantity* or as a `Vector<Double>`. If a quantity, the default units are the interanls one, but the output units can be selected as well.

5.3 Obtaining information from the MeasFrame

The frame can be used as an automatic converter. An example could be the automatic conversion to a sidereal time from a civil time (like UTC). All conversions that are used internally by the various conversion engines are available. Of course, the epoch should have been put into the frame for it to be converted; and in this case also the position should have been put into it. Available information from the frame:

```
// Get the epoch pointer (0 if not present)
const Measure *const epoch() const;
// Get the position pointer (0 if not present)
const Measure *const position() const;
// Get the direction pointer (0 if not present)
const Measure *const direction() const;
// Get the radial velocity pointer (0 if not present)
const Measure *const radialVelocity() const;
// Get the comet pointer (0 if not present)
const MeasComet *const comet() const;
// Get data from frame.
// Only available if appropriate measures are set,
// and the frame is in a calculating state.
// <group>
// Get TDB in days
Bool getTDB(Double &tdb);
// Get the longitude (in rad)
Bool getLong(Double &tdb);
// Get the latitude (in rad)
Bool getLat(Double &tdb);
// Get the position
Bool getITRF(MVPosition &tdb);
// Get the geocentric position (in m)
Bool getRadius(Double &tdb);
// Get the LAST (in days)
Bool getLAST(Double &tdb);
// Get the LAST (in rad)
Bool getLASTr(Double &tdb);
// Get J2000 coordinates (direction cosines)
Bool getJ2000(MVDirection &tdb);
// Get B1950 coordinates (direction cosines)
Bool getB1950(MVDirection &tdb);
// Get apparent coordinates (direction cosines)
Bool getApp(MVDirection &tdb);
// Get LSR radial velocity (m/s)
Bool getLSR(Double &tdb);
// Get the comet table reference type
Bool getCometType(uInt &tdb);
// Get the comet coordinates
Bool getComet(MVPosition &tdb);
// </group>
```

In the above reference is made to the *calculating state* of the frame. This state is set when the frame has been actively used by a `Measure::Convert` engine (i.e. have at least one *operator()* executed); or if you do it explicitly with: `MCFrame::make(frame);`.

A Appendix

A.1 Background literature

More information on the different astronomical conventions and data can be found in:

Time, Coordinates

- *Explanatory Supplement to the Astronomical Almanac*, P.K. Seidelman *et al.*, University Science Books, 1992
- *The Astronomical Almanac for the year yyyy*, London: The Stationary Office; Washington: U.S. Government Printing Office
- the "International Celestial Reference System" (*ICRS*)

Positions

- the *IERS* website

Frequency, Doppler, Velocity

- any astronomical textbook

Magnetic field

- the *IAGA* website

Baseline, uvw

- any textbook on radio astronomy

A.2 Details for individual Measures

In the following some comments are made per *Measure* type. A list is given of the known reference codes, and the frame information necessary to convert from/to each code is indicated.

A.2.1 MEpoch

An epoch in time. Internally maintained as an absolute time as a *Modified Julian Day* (or a *Greenwich Sidereal Date*) in two `Double` numbers. Formatting of an *MEpoch* (or any time expressed in time or angle units) can be done with the *MVTime* class.

MEpoch		
Code	*Description*	*Frame info*
LAST	Local Apparent Sidereal Time	MPosition
LMST	Local Mean Sidereal Time	MPosition
GMST1	Greenwich Mean ST1	
GAST	Greenwich Apparent ST	
UT1		
UT2		
UTC		
TAI		
TDT		
TCG		
TDB		
TCB		
IAT	$= TAI$	
GMST	$= GMST1$	MPosition
TT	$= TDT$	
UT	$= UT1$	
ET	$= TT$	
DEFAULT	$= UTC$	

A special code modifier `MEpoch::RAZE` exist. Its result is that *after* a conversion to a code with the `RAZE` bit set, the result will be truncated to integer days. Useful to find a sidereal time offset for a specific *UTC* date.

A.2.2 MPosition

A 3-dimensional vector, especially a position on (or rather *w.r.t.* Earth). Internally represented as a `Vector<Double>` with assumed units of m, independent of the constructing units. Normally given as a longitude, latitude and height (geodetic), or as a vector with origin in center of Earth.

The internal value of the `MPosition` class (i.e. *MVPosition*) is also the base class for the contents of the other 3-dimensional position and direction classes:

- MVDirection
- MVEarthMagnetic
- MVBaseline
- MVuvw

MPosition		
Code	*Description*	*Frame info*
ITRF	International Terrestrial Reference Frame	
WGS84	World Geodetic System	
DEFAULT	$= ITRF$	

A.2.3 MDirection

A 3-dimensional vector **of unit length**, indicating a direction, especially a direction in space (note that for the Solar system bodies the distance to the bodies is inherently known).

The solar system bodies' directions are *virtual* directions. They have no value until explicitly converted to a *real* coordinate system like $J2000$, $AZEL$, ...

MDirection		
Code	*Description*	*Frame info*
J2000	mean equator, equinox J2000.0	
JMEAN	mean equator, equinox epoch	MEpoch
JTRUE	true equator, equinox epoch	MEpoch
APP	apparent geocentric	MEpoch
B1950	mean equator, equinox B1950.0	
BMEAN	mean equator, equinox epoch	MEpoch
BTRUE	true equator, equinox epoch	MEpoch
GALACTIC	galactic coordinates	
HADEC	topocentric HA/Dec	MEpoch MPosition
AZEL	topocentric Az/El (N $\Rightarrow$ E)	MEpoch MPosition
AZELSW	topocentric Az/El (S $\Rightarrow$ W)	MEpoch MPosition
AZELNE	$= AZEL$	MEpoch
JNAT	geocentric natural frame	MEpoch
ECLIPTIC	ecliptic for J2000.0 equator, equinox	
MECLIPTIC	ecliptic for mean equator of date	MEpoch
TECLIPTIC	ecliptic for true equator of date	MEpoch
SUPERGAL	supergalactic coordinates	
ITRF	direction in ITRF Earth frame	MEpoch MPosition
TOPO	apparent topocentric	MEpoch
MERCURY	from JPL DE table	MEpoch
VENUS		MEpoch
MARS		MEpoch
JUPITER		MEpoch
SATURN		MEpoch
URANUS		MEpoch
NEPTUNE		MEpoch
PLUTO		MEpoch
SUN		MEpoch
MOON		MEpoch
COMET	any solar-system body	MEpoch Table[2]
DEFAULT	$= J2000$	

[2]The table needed contains a list of directions and distances as a function of MJD. The table can be generated from a standard formatted ascii text by the `measuresdata` Glish module. See there for how to generate a table. In future an XML driven parser could be used

A.2.4 MBaseline

A 3-dimensional vector, indicating a direction with a length (but no defined zero point). In principle they are identical to the *MDirection* directions. At the momemt the solar system bodies are not a valid baseline direction, but that could change.

MBaseline		
Code	*Description*	*Frame info*
J2000	mean equator, equinox J2000.0	
JMEAN	mean equator, equinox epoch	MEpoch
JTRUE	true equator, equinox epoch	MEpoch
APP	apparent geocentric	MEpoch
B1950	mean equator, equinox B1950.0	
BMEAN	mean equator, equinox epoch	MEpoch
BTRUE	true equator, equinox epoch	MEpoch
GALACTIC	galactic coordinates	
HADEC	topocentric HA/Dec	MEpoch MPosition
AZEL	topocentric Az/El (N $\Rightarrow$ E)	MEpoch MPosition
AZELSW	topocentric Az/El (S $\Rightarrow$ W)	MEpoch MPosition
AZELNE	$= AZEL$	MEpoch
JNAT	geocentric natural frame	MEpoch
ECLIPTIC	ecliptic for J2000.0 equator, equinox	
MECLIPTIC	ecliptic for mean equator of date	MEpoch
TECLIPTIC	ecliptic for true equator of date	MEpoch
SUPERGAL	supergalactic coordinates	
ITRF	direction in ITRF Earth frame	MEpoch MPosition
DEFAULT	$= ITRF$	

A.2.5 Muvw

A 3-dimensional vector, indicating a UVW-coordinate.

Muvw		
Code	*Description*	*Frame info*
J2000	mean equator, equinox J2000.0	
JMEAN	mean equator, equinox epoch	MEpoch
JTRUE	true equator, equinox epoch	MEpoch
APP	apparent geocentric	MEpoch
B1950	mean equator, equinox B1950.0	
BMEAN	mean equator, equinox epoch	MEpoch
BTRUE	true equator, equinox epoch	MEpoch
GALACTIC	galactic coordinates	
HADEC	topocentric HA/Dec	MEpoch MPosition
AZEL	topocentric Az/El (N $\Rightarrow$ E)	MEpoch MPosition
AZELSW	topocentric Az/El (S $\Rightarrow$ W)	MEpoch MPosition
AZELNE	$= AZEL$	MEpoch
JNAT	geocentric natural frame	MEpoch
ECLIPTIC	ecliptic for J2000.0 equator, equinox	
MECLIPTIC	ecliptic for mean equator of date	MEpoch
TECLIPTIC	ecliptic for true equator of date	MEpoch
SUPERGAL	supergalactic coordinates	
ITRF	direction in ITRF Earth frame	MEpoch MPosition
DEFAULT	$= ITRF$	

A.2.6 MEarthMagnetic

A 3-dimensional vector: the value of the Earth' magnetic field. The model used to calculate the field is the International Geomagnetic Reference Field. The *IGRF* field is a *virtual* field, without any value. It obtains its value after an explicit conversion to a normal, real coordinate system.

MEarthMagnetic		
Code	*Description*	*Frame info*
IGRF	the international reference field model	MEpoch MPosition
J2000	mean equator, equinox J2000.0	
JMEAN	mean equator, equinox epoch	MEpoch
JTRUE	true equator, equinox epoch	MEpoch
APP	apparent geocentric	MEpoch
B1950	mean equator, equinox B1950.0	
BMEAN	mean equator, equinox epoch	MEpoch
BTRUE	true equator, equinox epoch	MEpoch
GALACTIC	galactic coordinates	
HADEC	topocentric HA/Dec	MEpoch MPosition
AZEL	topocentric Az/El (N $\Rightarrow$ E)	MEpoch MPosition
AZELSW	topocentric Az/El (S $\Rightarrow$ W)	MEpoch MPosition
AZELNE	$= AZEL$	MEpoch
JNAT	geocentric natural frame	MEpoch
ECLIPTIC	ecliptic for J2000.0 equator, equinox	
MECLIPTIC	ecliptic for mean equator of date	MEpoch
TECLIPTIC	ecliptic for true equator of date	MEpoch
SUPERGAL	supergalactic coordinates	
ITRF	direction in ITRF Earth frame	MEpoch MPosition
DEFAULT	$= ITRF$	

A.2.7 MFrequency

The `MFrequency` class describes the characteristics of an electro-magnetic wave. Internally the value is in Hz, but an *MFrequency* object can be constructed from any characteristic that is understood (period, frequency, angular frequency, wavelength, wave number, energy, momentum).

The *MFrequency* class has special methods to convert the frequency from an *MDoppler* object, if a rest frequency is known.

MFrequency		
Code	*Description*	*Frame info*
REST	spectral line rest frequency	MDirection MRadialVelocity
LSR	dynamic local standard of rest	MDirection
LSRK	kinematic local standard of rest	MDirection
BARY	barycentric frequency	MDirection
GEO	geocentric frequency	MDirection MEpoch
TOPO	topocentric frequency	MDirection MEpoch MPosition
GALACTO	galactocentric frequency	MDirection
DEFAULT	= *LSR*	

A.2.8 MRadialVelocity

The `MRadialVelocity` class describes the radial velocity of an astronomical object. Internally the value is in m/s. Methods are available (if the rest frequency of a spectral line is known), to convert the velocity to a frequency. Also, the radial velocity can be derived from an *MDoppler* object, which has the radial velocity in the often better known redshift or radio-astronomical Doppler shift.

MRadialVelocity		
Code	*Description*	*Frame info*
LSR	dynamic local standard of rest	MDirection
LSRK	kinematic local standard of rest	MDirection
BARY	barycentric frequency	MDirection
GEO	geocentric frequency	MDirection MEpoch
TOPO	topocentric frequency	MDirection MEpoch MPosition
GALACTO	galactocentric frequency	MDirection
DEFAULT	= *LSR*	

A.2.9 MDoppler

The `MDoppler` class describes the radial velocity of an astronomical object in a variety of special astronomical "Doppler" shifts. It can be generated from quasi velocities; or from a dimensionless quantity, interpreted as a fraction of the velocity of light.

Conversion to a "real" radial velocity is possible with this class. In the following F stands for ν/ν_0, where ν_0 is the rest frequency.

MDoppler		
Code	*Description*	*Frame info*
RADIO	radio definition: $1 - F$	
Z	redshift: $-1 + 1/F$	
RATIO	frequency ratio: F	
BETA	relativistic: $(1 - F^2)/(1 + F^2)$	
GAMMA	$(1 + F^2)/2F$	
OPTICAL	$= Z$	
RELATIVISTIC	$= BETA$	
DEFAULT	$= RADIO$	