

Windows Astronomical Resource Protocol (WARP)

Implementation Specification

Copyright 2004 Astrometric Instruments, Inc.

last revision date: 11-Dec-2004

This document can be ordered as Astrometric Instruments' part DOC-07

Windows Astronomical Resource Protocol

Revision

Covered in this document: WARP version 1.10.000 11-Dec-2004.

Revision history is included in the separate document entitled WARP_RevisionHistory.pdf.

Premise

WARP is a standard that defines a straightforward mechanism for convenient communications between “client” astronomy software (e.g. planetarium software, observer assistant software: referred to as “Client” in this document) and “resource Provider” software (e.g. telescope control, CCD camera control software: referred to as the “Provider” in this document) running under windows on the same PC.

WARP is designed to be a “fast path” to implementing inter-process communications between astronomical software in the windows environment.

WARP: not the only astronomical inter-software communications method

In late 1998, we began looking for a means to interface SkyGuide-Win to other windows-based astronomical software. At the time we found only one existing method: SkyMap PRO's “scope driver” approach. Shortly afterwards (early 1999), Software Bisque released their “TeleAPI” approach. Both of these approaches are quite workable. Not to their discredit however, they are specific to one manufacturer of software.

Also in late 1998, Bob Denny introduced ASCOM (<http://ASCOM-Standards.org>). ASCOM proposes a “component-based” approach to providing astronomical software functionality under windows. With ASCOM one can get their telescope control software from one company, CCD control software from another and planetarium software from a third, etc. The goal of ASCOM is strongly endorsed by Astrometric Instruments.

There are other astronomical software “interfaces” that exist: Sienna Soft's Starry Night “Plug-in” method and the Digital Sky Voice “link bridges” are just two examples of effective and capable interfaces. There may be several others that the authors are not presently aware of.

All of these methods are quite functional and do work fine!

Why then WARP?

WARP is designed to be straightforward to use. It is designed with the idea that authors of Windows-based astronomical software can make use of a quick and easy approach to providing inter-software communications. A developer can take the WARP standard, along with example Client and Provider applications, and implement an interface in very little time.

WARP is also designed to be vendor-independent. Sure, AI is proposing the standard but our reason for doing this is exactly why we suspect others will be interested: we need a quick/easy way to implement an interface to other windows-based software applications and want it to be quick/easy for authors of other programs to interface to our software. And once the interface is complete we want it to work with as many other applications as possible.

WARP is not meant to replace other methods, in particular ASCOM. ASCOM has several advantages over WARP that may/may-not be important in specific situations:

- Uses a COM-based approach. The advantages are that one program can effectively “call into” another program. With this capability one program can use a significant “component” of another program. For example, a “observing planner” program could have a planetarium program draw a star map for it.
- Provides for remote access. Windows COM-based applications can not only communicate on the same PC but can also be communicated over a network.

Astrometric Instrument's interest

Astrometric Instrument's general interest in this standard is stated in the premise above. Astrometric Instrument's specific interest is in providing communications from planetarium and observer assistant software to our newer Maestro SkyWalker-interface software and our older SkyGuide-Win telescope control system software.

The WARP standard

Strict requirements

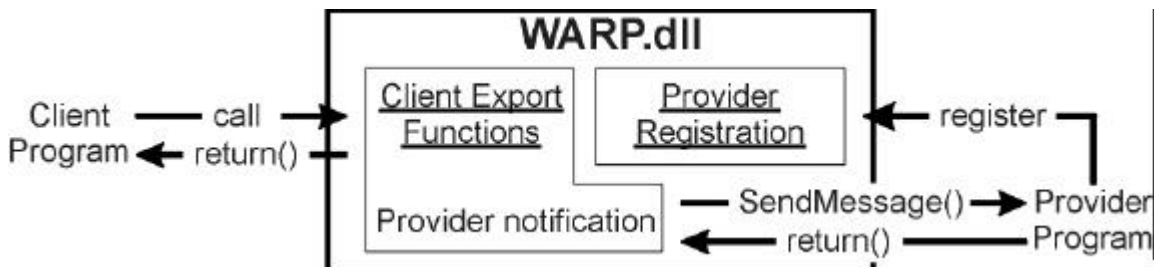
The foremost requirement of this proposed standard is that it is completely backwards compatible with older versions of the standard. This will assure that any Client or Provider software that works now will continue to work with future versions of WARP. As such, a version standard is put forth where any version starting with “0.” is a developmental version and subject to change (e.g. version 0.10). From version 1.00 on-wards the backwards compatible “dictum” is adhered to.

The way WARP works

A Windows DLL (WARP.dll) provides an interface between “Client” and “Provider” software. When a Client program wants to interface to a Provider it loads WARP.dll (which resides in the \Program Files\Common Files\System folder). The Client can then immediately start to call “export functions” within WARP.dll to communicate with any currently active Providers.

Note: WARP.dll is installed in the \Program Files\Common Files\System folder whenever a WARP Provider application is installed on the PC. If WARP.dll does not exist on the PC then it can be assumed that there is no WARP Provider installed.

Graphically, the connection between Client, WARP.dll and Provider looks as follows:



Client software initiates all action. For example: a Client may want to know the apparent celestial coordinates that a telescope is presently pointed at. It would use the `TCS_GetCelestialCoords()` export function. The sequence that would transpire is as follows:

1. Client calls `TCS_GetCelestialCoords()` inside `WARP.dll`.
2. `WARP.dll` notifies the associated Provider (via `SendMessage()` Win32 function) of the request for celestial position.
3. The Provider, in its windows messaging loop, loads the current Apparent celestial position at the pointer sent from `WARP.dll` and returns.
4. `WARP.dll` then loads the current Apparent celestial position at the pointer provided by the Client and returns control to the Client.

It is important that `WARP.dll` and Provider code execute quickly since they block execution in the Client during the above sequence. Note: `WARP.dll` is written as a “go between” the Client and Provider so that neither requires access to memory locations within the others address space (which, of course, is illegal under Windows).

WARP Export Function reference

The WARP.dll file provides a rich assortment of export functions. Client and Provider software use these export functions to communicate.

WARP uses “explicit linking”. With explicit linking, the client executable using the DLL must make function calls to explicitly load and unload the DLL, and to access the DLL's exported functions. The client executable must also call the exported functions through a function pointer.

Why use explicit linking? Explicit linking eliminates the need to link the application with an import library. If changes in the DLL cause the export ordinals to change, applications using explicit linking do not have to re-link (assuming they are calling GetProcAddress with a name of a function and not with an ordinal value), whereas applications using implicit linking must re-link to the new import library.

Note: the WARP.def file is not provided as part of the WARP distribution since Explicit linking is used.

The WARP_ClientExample program, provided by Astrometric Instruments, shows how to use explicit linking.

WARP typedefs, definitions and macros are included in the header file WARP.h. A few notables:

- ◆ The WARP “coordinates” structure:

```
struct WARP_Coords
{
    double X;      // Radians
    double Y;      // Radians
};
```

- ◆ WARP constants: several are defined for general use. An example is “W_ON”. All WARP constants are preceded with “W_” to minimize the potential for double assignment within any given program. WARP constants are defined in the header file WARP.h.

Function prototype format: short __stdcall g_xxx(aaa *p_bbb, <ccc ddd>,...);

- ◆ All WARP functions use the __stdcall calling convention. This is the allows for load/use of WARP.dll from C and Visual Basic programs.
- ◆ “return_value” indicates success or failure of the request (i.e. function call). The complete list is contained in the “WARP Client export function return values” section of the WARP header file (WARP.h). Some examples include:
 - WARP_ACK... WARP.dll did not encounter a problem in processing the call.
 - WARP_NACK... WARP.dll **did** encounter a problem in processing the call.
 - WARP_PROVIDER_UNAVAILABLE ... an appropriate Provider was not available (i.e. not “registered”).
 - WARP_PROVIDER_DOES_NOT_SUPPORT... the appropriate Provider does not support the Client request.

- W_TCS_PARAMETER1_ERROR... parameter #1 is in error (e.g. contains a value that is out of bounds).
- W_TCS_PARAMETER2_ERROR... parameter #2 is in error (e.g. contains a value that is out of bounds).
- ◆ Function name: “g_xxx”: “g” is the group and “xxx” the name. For example, TCS_GetAlignmentStatus() is the function, from the TCS (Telescope Control System) group, used to get the alignment status of the TCS Provider.
- ◆ Parameters: “aaa” and “ccc” are types and “p_bbb” and “ddd” are parameters (“p_bbb” is a pointer parameter). Note: if a function does not return WARP_ACK then the value referred to by a pointer may be invalid.

Client Export Functions

Telescope Control System (TCS) group:

Status

- ◆ short __stdcall **TCS_GetVersion**(char *ver_string, str_len). Loads the WARP TCS version into *ver_string. Format is M.NN.SSS. M = major version, NN = minor version, and SSS = subversion. The str_len parameter specifies the maximum number of characters (including the NULL termination character) that should be copied into the string.
- ◆ short __stdcall **TCS_GetMessageLevel**(short *p_level). Loads the current “message level” (see TCS_SetMessageLevel() for complete details) into *p_level.
- ◆ Short __stdcall **TCS_GetLastMessage**(char *p_message, short str_len). Loads the last message from the Provider into *p_message. The str_len parameter specifies the maximum number of characters (including the NULL termination character) that should be copied into the string.

This function is the only means through which Client software can get the messages (e.g. status, warnings, alerts, fatal errors, etc.) that the Provider issues to its interactive users. See TCS_SetMessageLevel() for more/complete details

- ◆ short __stdcall **TCS_GetControlStatus**(short *p_status). Loads one of the following into *p_status:
 - W_TCS_INACTIVE: if not yet “controlling” the telescope (e.g. Maestro when it has not yet connected to SkyWalker).
 - W_TCS_ACTIVE: if “controlling” the telescope (e.g. Maestro once it has connected to SkyWalker).
- ◆ short __stdcall **TCS_GetAlignmentStatus**(short *p_status). Loads one of the following into *p_status:
 - W_TCS_NOT_ALIGNED
 - W_TCS_CELESTIAL_ALIGNED
 - W_TCS_CELESTIAL_AND_EARTH_ALIGNED

- ◆ short __stdcall **TCS_GetCelestialCoords**(struct WARP_Coords *p_coords). If the TCS Provider is celestial aligned then the WARP_Coords structure *p_coords is loaded with radian RA (X) and Dec (Y) and the return value is WARP_ACK. Otherwise the return value is W_TCS_NOT_ALIGNED. Note: the RA/Dec values are subject to the current TCS Provider's Refraction correction settings. Specifically, if Refraction correction is enabled then the coordinates represent the "Apparent position" that the telescope is viewing. The Apparent position is the real telescope axial position corrected "downwards" to arrive at the celestial position that the telescope is apparently viewing. Note: the coordinates are always for the "Epoch of date".
- ◆ short __stdcall **TCS_GetEarthCoords**(struct WARP_Coords *p_coords). If the TCS Provider is not Earth aligned then the return value is W_TCS_NOT_EARTH_ALIGNED (and cords are invalid). If the TCS Provider is not Celestial aligned then the return value is W_TCS_NOT_ALIGNED (and cords are also invalid). Otherwise, the WARP_Coords structure *p_coords is loaded with radian Topocentric Azimuth (X) and Topocentric Altitude (Y) and the return value is WARP_ACK. Note: per standard astronomical convention Azimuth is "South-referenced" (i.e. 0.0 at South) and positive West/clockwise.
- ◆ short __stdcall **TCS_GetSiderealTime**(double *p_sidtime). If the TCS Provider is Earth aligned then the double value *p_sidtime is loaded with the Provider's Local Sidereal Time as a fraction of a sidereal day (i.e. range of 0.0 to 1.0). Otherwise the return value is W_TCS_NOT_EARTH_ALIGNED.
- ◆ short __stdcall **TCS_GetGoToProgress**(short *p_precent_left). Loads *p_precent_left with a short integer representing the percentage remaining of the most recent GoTo (0 to 100).
- ◆ short __stdcall **TCS_GetScopeOrientation**(short *p_orientation, short *p_direction). If the TCS Provider is celestial aligned then:
 - *p_orientation is loaded with the "side" of the mount that the telescope is currently pointing (i.e. "Scope Orientation"). Possible values are W_TCS_PRIMARY (i.e. telescope is currently on the same "side" as celestial aligned) or W_TCS_SECONDARY (i.e. telescope is on the opposite "side" as celestial aligned).
 - ~~*p_direction is valid only when the TCS Provider is controlling a telescope that operates "in the East" or "in the West" with respect to the celestial meridian (e.g. German equatorial). If this is the case *p_direction returns W_TCS_EAST_SIDE or W_TCS_WEST_SIDE as appropriate.~~

If the TCS Provider is not celestial aligned the return value is W_TCS_NOT_ALIGNED.
- ◆ short __stdcall **TCS_GetControlSettingShort**(short setting_specifier, short *p_setting_value). This function is used to poll the TCS Provider for the current value of a short internal setting. The following settings, and associated setting_specifiers, are supported (note the value of the setting is placed in the location pointed to by p_setting_value):

- **W_TCS_TRATE**: current track rate of the TCS Provider. See the short `__stdcall TCS_SelectTrackRate(short track_rate)` function for track rate choices/definitions.
- **W_TCS_PARK_MARK**: returns **W_YES** if the park position is marked and **W_NO** if it is not.
- **W_TCS_SETTING_REFRACTION_CORR_ENABLED**: status of the TCS Provider's refraction correction (**W_ON** or **W_OFF**).

If the `setting_specifier` parameter specifies an invalid setting the function returns **W_TCS_PARAMETER1_ERROR** otherwise the function returns **WARP_ACK**.

- ◆ short `__stdcall TCS_GetControlSettingDouble(short setting_specifier, double *p_setting_value)`. This function is used to poll the TCS Provider for the current value of a double internal setting. The following settings, and associated `setting_specifiers`, are supported (note the value of the setting is placed in the location pointed to by `p_setting_value`):

- **W_TCS_SETTING_INPUT_EPOCH**: the “Input Epoch” that the TCS Provider currently uses to convert incoming coordinates to Apparent coordinates (e.g. 2000.0). A value of 0.0 indicates that the TCS Provider uses “Epoch of date” (i.e. no corrections are applied to incoming coordinates).
- **W_TCS_SETTING_REFRACTION_CORR_TEMP**: the atmospheric temperature, in degrees Celsius, that the TCS Provider is using for refraction correction.
- **W_TCS_SETTING_REFRACTION_CORR_PRESSURE** the atmospheric pressure, in millibars, which the TCS Provider is using for refraction correction.
- **W_TCS_SETTING_SITE_LATITUDE** the latitude of the current TCS Provider site in radians (positive North).
- **W_TCS_SETTING_SITE_LONGITUDE** the longitude of the current TCS Provider site in radians (positive East).

If the `setting_specifier` parameter specifies an invalid setting the function returns **W_TCS_PARAMETER1_ERROR**. Otherwise, the function returns **WARP_ACK** unless the Provider is not Earth aligned for the **W_TCS_SETTING_SITE_LATITUDE** or **W_TCS_SETTING_SITE_LONGITUDE** `setting_specifier` values (in which case **W_TCS_NOT_EARTH_ALIGNED** is returned).

- ◆ short `__stdcall TCS_GetPeripheralOutput(short peripheral_number, short *p_state)`. Loads `*p_state` with the current state of the TCS Provider's peripheral output as specified by `peripheral_number`. Possible values for `*p_state` are:

- **W_ON** if the specified peripheral is “on”.
- **W_OFF** if the specified peripheral is “off”.
- **W_UNDEFINED** if the state of the specified peripheral is undefined or unknown.

If the specified `peripheral_number` is not supported by the TCS Provider then the function returns **W_TCS_PARAMETER1_ERROR**. Otherwise, the function returns **WARP_ACK**.

Settings

- ◆ short `__stdcall TCS_SetInteractiveDialog(short state)`. Set state to `W_OFF` to disable all interactive dialogs that the TCS Provider would normally make with the user (e.g. pop-up windows that request information from the user). Interactive dialogs that the Provider should disabled are those which could prevent smooth remote operation. The Provider will have to take reasonable default actions with interactive dialogs disabled. These defaults are documented in the TCS Provider's documentation. Set to `W_ON` to enable interactive pop-up windows.
- ◆ short `__stdcall TCS_SetMessageLevel(short level)`. WARP provides a feature where a Provider can “post” a message to Clients. The Provider does this by calling the `TCS_ProviderPostMessage(char *p_message)` function. This function loads a global message buffer inside `WARP.dll` with the message pointed to by `p_message`. The Provider will only post messages that are at or above the current message level. The current message level is set by the Client with the `TCS_SetMessageLevel()` function as follows:

Message Level	Result
1	All messages get posted.
2	e.g. when Maestro is the TCS Provider, only Warnings and Alerts are posted.
3	e.g. when Maestro is the TCS Provider, only Alerts are posted.
4	Only fatal errors get posted.

Note: the Provider’s default message level should be 4.

- ◆ short `__stdcall TCS_SetControlMode(short state)`. Provides a means for Client software to set the TCS Provider “control status” to `W_TCS_ACTIVE` or `W_TCS_INACTIVE`:
 - To cause the Provider to begin “controlling” the telescope (e.g. connect to the control system hardware) set state to `W_TCS_ACTIVE`. For SkyGuide, this is equivalent to entering *Land mode*.
 - To cause the Provider to cease “controlling” the telescope (e.g. disconnect from the control system hardware) set state to `W_TCS_INACTIVE`. For SkyGuide this is equivalent to leaving *Land mode* and returning to *StartUp mode*.

Note: some Provider software may not support this function. For example, this function is not supported by Maestro since the connection state of Maestro/SkyWalker is not controllable (i.e. Maestro **always** looks for, and tries to connect to, SkyWalker).
- ◆ short `__stdcall TCS_SelectTrackRate(short track_rate)`. Sets the TCS Provider’s current rate of tracking. The choices and constant values to use for `track_rate` are as follows:
 - For Sidereal track rate: `W_TCS_TRATE_SIDEREAL`
 - For Lunar track rate: `W_TCS_TRATE_LUNAR`
 - For Solar track rate: `W_TCS_TRATE_SOLAR`

- For Custom track rate: W_TCS_TRATE_CUSTOM
- For Drift (i.e. no tracking): W_TCS_TRATE_DRIFT
- ◆ short __stdcall **TCS_SetCelestialAlignPoint**(WARP_Coords coords, short number, short side). Specifies the current Catalog celestial coordinates that the telescope is pointing. The TCS Provider is responsible for correcting the Catalog coordinates from the Input Epoch to the “Epoch of date” and is responsible for correcting for the effects of atmospheric refraction (proper Input Epoch and refraction parameters should be set with other TCS export functions).

The TCS_SetCelestialAlignFromAlignPoints() function must be called to complete the Celestial alignment.

The number of alignment points necessary, before completing celestial alignment, depends on the specifics of the TCS. For example, only one alignment point will be necessary if the TCS is setup to allow one-star celestial alignment. No alignment points are necessary if the TCS provides “align from last position” capability.

~~A reference (e.g. handle) to the alignment point is provided through the **number** parameter. If this parameter is the same as an existing alignment point then the existing point is over-written. WARP currently allows only one alignment point and the **number** parameter is ignored and subsequent sighting entries simply over-write the one point.~~

~~For Asymmetric mounts (e.g. German Equatorials) the side of the *Meridian Limits* that the telescope is currently pointing (i.e. W_TCS_EAST_SIDE or W_TCS_WEST_SIDE) must be specified. If you want the TCS to calculate (from Sidereal time, coords and an understanding of the telescope *Meridian Limits*) the side then specify W_TCS_UNSPECIFIED_SIDE for the **side** parameter. This will be the situation for all mount types except Asymmetric mounts and may even be used for Asymmetric mounts when the Client is certain that the telescope is pointed well outside of the *Meridian Avoidance* or overlap zone.~~

The following return values are provided:

- W_WARP_ACK: the alignment point was accepted.
- W_TCS_NOT_IN_LAND_MODE: this function can only be used in *Land mode*.
- ~~• W_TCS_COORDS_BELOW_ALT_LIMIT: the TCS may impose an altitude limit for the provided coordinates. This return value results if the coords are below this limit.~~
- ~~• W_TCS_COORDS_BEYOND_MERIDIAN_LIMIT: the coords are beyond the *Meridian Limits*. This alignment point is discarded. Note: this is only an error if W_TCS_UNSPECIFIED_SIDE is passed in the side parameter.~~
- ~~• W_TCS_COORDS_ABOVE_POLE_LIMIT: the coords are above the *ScopePole limit*. This alignment point is discarded.~~
- ~~• W_TCS_COORDS_BELOW_DOWN_LIMIT: the coords are below the *Down Limit*. This alignment point is discarded.~~
- W_TCS_EXCEEDED_ALIGNMENT_POINT_LIMIT:

- ◆ short __stdcall **TCS_RemoveCelestialAlignPoint**(short number). TBD.
- ◆ short __stdcall **TCS_SetCelestialAlignFromAlignPoints**(void). If a sufficient number of celestial alignment points have been set with the `TCS_SetCelestialAlignPoint()` function then the `TCS_SetCelestialAlignFromAlignPoints()` function will complete the celestial alignment.

The last alignment point is taken as the current Catalog celestial coordinate. Possible return values include all the values that the `TCS_SetCelestialAlignPoint()` function returns with, with the addition of `W_TCS_TOO_FEW_ALIGNMENT_POINTS` if an insufficient number of alignment points have been recorded. Note: return values that address limit violations refer to the position of the last alignment point.

This function can be used to align from the “last” known position that the telescope was pointing **if** the TCS Provider provides “align from last” capability. To accomplish an “align from last” simply call the `TCS_SetCelestialAlignFromAlignPoints()` function without any previous calls to `TCS_SetCelestialAlignPoint()`.

- ◆ short __stdcall **TCS_SetCelestialAlignFromEarthCoords**(WARP_Coords coords, short side). This function completes Celestial alignment from Earth (i.e. Alt/Az) coordinates. The TCS Provider must determine the Celestial coordinates (to align from) corresponding to the Alt/Az provided.

The `WARP_Coords` structure `coords` must be loaded with radian Topocentric Azimuth (X) and Topocentric Altitude (Y). Note: per standard astronomical convention Azimuth is “South-referenced” (i.e. 0.0 at South) and positive West/clockwise.

~~For Asymmetric mounts (e.g. German Equatorials) the side of the *Meridian Limits* that the telescope is currently pointing (i.e. `W_TCS_EAST_SIDE` or `W_TCS_WEST_SIDE`) must be specified.~~ If you want the TCS to calculate (from Sidereal time, `coords` and an understanding of the telescope *Meridian Limits*) the side then specify `W_TCS_UNSPECIFIED_SIDE` for the **side** parameter. This will be the situation for all mount types except Asymmetric mounts and may even be used for Asymmetric mounts when the Client is certain that the telescope is pointed well outside of the *Meridian Avoidance* or overlap zone.

The following return values are provided:

- `W_WARP_ACK`: the alignment point was accepted.
- `W_TCS_NOT_IN_LAND_MODE`: this function can only be used in *Land mode*.
- If the TCS Provider is not Earth aligned then the return value is `W_TCS_NOT_EARTH_ALIGNED` and the alignment is not completed.
- ◆ short __stdcall **TCS_VoidCelestialAlign**(void). If the TCS is in *Celestial mode* then it is promptly returned to *Land mode* and `WARP_ACK` is returned. If the TCS is in any other mode this function has no effect and `WARP_NACK` is returned.
- ◆ short __stdcall **TCS_Calibrate**(WARP_Coords coords). If the TCS Provider is celestial aligned then the TCS celestial coordinates are calibrated to `coords`. As with alignment, the TCS Provider is responsible for correcting the Catalog coordinates from

the Input Epoch to the “Epoch of date” and is responsible for correcting for the effects of atmospheric refraction (proper Input Epoch and refraction parameters should be set with other TCS export functions).

If the Calibration was illegal, because it was beyond telescope limits, then the return value is WARP_NACK.

If the TCS Provider is not celestial aligned the return value is W_TCS_NOT_ALIGNED. Otherwise the return value is WARP_ACK.

- ◆ short __stdcall **TCS_CalibrateRA_Dec** (double ra, double dec). Provides the same function as TCS_Calibrate() however the parameters are individual RA/Dec rather than a WARP_Coords structure. This function is provided for use with Visual Basic (VB can not pass “user defined data types”, UDTs, to DLLs).
- ◆ short __stdcall **TCS_CalibrateFromEncoders**(void). Signals to the Provider that it should calibrate its current position from axial encoders.
- ◆ short __stdcall **TCS_SetControlSettingDouble**(short setting_specifier, double value). The behavior of this function depends on the setting_specifier:
 - **W_TCS_SETTING_INPUT_EPOCH**: Sets the Epoch that the TCS Provider uses to convert the Client’s catalog coordinates to apparent coordinates (i.e. compensating for Precession, Aberration, Nutation, etc.). The specified epoch value is a real number (e.g. 2000.0). A value of 0.0 indicates that the TCS Provider should use “Epoch of date” (i.e. no corrections are applied to incoming coordinates).
If the epoch parameter is out of range (allowable range is 1950.0 to 2050.0 and 0.0) the function returns W_TCS_PARAMETER2_ERROR otherwise the function returns WARP_ACK.
 - **W_TCS_SETTING_REFRACTION_CORR_TEMP**: Sets the atmospheric temperature that the TCS Provider uses to correct for atmospheric refraction. Units are degrees Celsius.
If the temperature value is out of range (allowable range is -70.0 to 70.0) the function returns W_TCS_PARAMETER2_ERROR otherwise the function returns WARP_ACK.
 - **W_TCS_SETTING_REFRACTION_CORR_PRESSURE**: Sets the atmospheric pressure that the TCS Provider uses to correct for atmospheric refraction. Units are millibars.
If the pressure parameter is out of range (allowable range is 100.0 to 1500.0) the function returns W_TCS_PARAMETER2_ERROR otherwise the function returns WARP_ACK.
 - **W_TCS_SETTING_SITE_LATITUDE**: Sets the latitude for the currently selected site. The value is in radians (positive North). If the value is out of range the function returns W_TCS_PARAMETER2_ERROR. If the Provider encounters any other problems (e.g. a site is not selected in the Provider or the Provider is in an improper mode to accept changes to latitude) then the function returns WARP_NACK.

- **W_TCS_SETTING_SITE_LONGITUDE**: Sets the longitude for the currently selected site. The value is in radians (positive East). If the value is out of range the function returns **W_TCS_PARAMETER2_ERROR**. If the Provider encounters any other problems (e.g. a site is not selected in the Provider or the Provider is in an improper mode to accept changes to latitude) then the function returns **WARP_NACK**.
- ◆ short __stdcall **TCS_SetRefractionCorr**(short state). Sets the TCS Provider's refraction correction on or off. Possible values for state are **W_ON** or **W_OFF**.
If the state parameter is not **W_ON** or **W_OFF** the function returns **W_TCS_PARAMETER1_ERROR**, otherwise the function returns **WARP_ACK**.
- ◆ short __stdcall **TCS_SetParkMark** (void). “Marks” the telescope's current azimuth and altitude as the park position. The TCS Provider must be celestial aligned. If not celestial aligned, the TCS Provider returns **W_TCS_NOT_ALIGNED**, otherwise it returns **WARP_ACK**.
- ◆ short __stdcall **TCS_SetPeripheralOutput**(short peripheral_number, short state). Sets the state of the TCS Provider's peripheral output as specified by peripheral_number. Possible values for state are:
 - **W_ON** to set the specified peripheral “on”.
 - **W_OFF** to set the specified peripheral “off”.If the specified peripheral_number is not supported by the TCS Provider then the function returns **W_TCS_PARAMETER1_ERROR**. Otherwise, the function returns **WARP_ACK**.

Actions

- ◆ short __stdcall **TCS_KillMotion**(void). Terminates all non-tracking motion in the TCS Provider.
- ◆ short __stdcall **TCS_GoToCelestialCoords**(WARP_Coords coords). If the TCS Provider is celestial aligned then the TCS will start a GoTo the Catalog celestial coordinates specified in coords. The TCS Provider is responsible for correcting the Catalog coordinates from the Input Epoch to the “Epoch of date” and is responsible for correcting for the effects of atmospheric refraction (proper Input Epoch and refraction parameters should be set with other TCS export functions).
~~If the GoTo was illegal then the return value is WARP_NACK.~~ If the TCS Provider is not celestial aligned the return value is **W_TCS_NOT_ALIGNED**. Otherwise the return value is **WARP_ACK**.
- ◆ short __stdcall **TCS_GoToCelestialRA_Dec**(double ra, double dec). Provides the same function as **TCS_GoToCelestialCoords**() however the parameters are individual RA/Dec rather than a **WARP_Coords** structure. This function is provided for use with Visual Basic (VB can not pass “user defined data types”, UDTs, to DLLs).
- ◆ short __stdcall **TCS_GoToEarthCoords**(WARP_Coords coords). If the TCS Provider is celestial aligned and Earth aligned then the TCS will start a GoTo the Earth

(i.e. Alt/Az) coordinates specified in coords. Coords must be set with the target Topocentric Azimuth (X) and Topocentric Altitude (Y) in radians. Note: per standard astronomical convention Azimuth is “South-referenced” (i.e. 0.0 at South) and positive West/clockwise.

After the GoTo is complete, the TCS Provider will be in a Track Rate of Drift (i.e. the TCS Provider assumes that since the target is an Alt/Az position that tracking should be turned off).

If the TCS Provider is not celestial aligned the return value is W_TCS_NOT_ALIGNED. If the TCS Provider is not Earth aligned the return value is W_TCS_NOT_EARTH_ALIGNED. Otherwise the return value is WARP_ACK.

- ◆ short __stdcall **TCS_GoToPark**(void). If the TCS Provider is celestial aligned then the TCS will start a GoTo the previously marked park position. If not celestial aligned, the GoTo will not occur and the return value is W_TCS_NOT_ALIGNED. If the park position has not yet been marked then the GoTo will not occur and the return value is W_TCS_NO_PARK_MARK, otherwise the return value will is WARP_ACK.
- ◆ short __stdcall **TCS_StickyMotion**(short direction, short state). If the state is T_ON, this function causes the Provider to begin motion (at its currently selected rate) in the specified direction (i.e. W_TCS_UP, W_TCS_DOWN, W_TCS_LEFT or W_TCS_RIGHT). If the state is not T_ON (e.g. T_OFF) then motion in the specified direction is stopped. To stop all motion use the TCS_KillMotion() function.

Provider Export Functions

Telescope Control System (TCS) group:

Administrative

- ◆ short __stdcall **TCS_RegisterProvider**(HWND handle, short version). Provider “registers” itself as the TCS Provider and passes the handle to its main window to WARP.dll. Also, declares, to WARP.dll, what version of WARP the Provider supports. WARP.dll will present a Provider interface compatible with that version. The return value is always WARP_ACK. ~~If another Provider is already registered it is notified (via a TBD message) and then loses registration.~~
- ◆ short __stdcall **TCS_ProviderPostMessage**(char *p_message). This function provides the Provider with a means of “posting” messages to WARP.dll. Note: the Provider should only post message types at a “level” consistent with the message level set with the TCS_SetMessageLevel(short level) function. Note: the Provider’s default message level should be 4.

For use with Provider software written in Microsoft Visual Basic

The WARP interface (contained in WARP.dll) provides a communications path between the client and the Provider. After receiving a client request, WARP.dll uses the Windows SendMessage function to send the request type and associated parameters to the Provider. In several instances the parameters include a memory address pointer to a data location within WARP.dll data memory shared by all users of the dynamic link library. This memory address

pointer may point to data to be transferred to the Provider (from the client) or to be updated by Provider and returned to the client. Because Visual Basic does not support retrieving or storing data based on pointers, support functionality is required to synthesize data retrieves based on pointers and data storage based on pointers.

Retrieve Support Functions

Retrieve support functions are provided within WARP.dll that can be called by a Visual Basic program passing the address of the data to WARP.dll. The support routine retrieves the data from the specified memory location and returns the data to the calling program. Specific support routines are provided to retrieve short and double data types. Additionally, a support routine is provided to retrieve coordinate parameter values from a WARP coordinate structure.

WARP_SHORTRETRIEVEEXPORT TCS_ProviderSupportRetrieveShort(short *p_SupportShortVal)

Description: When the warp command requires the Provider to fetch a short value, a Visual Basic program can call TCS_ProviderSupportRetrieveShort passing the desired address of the short value. This address was provided in the message sent by WARP.dll to the Provider. This warp support routine will fetch the short value from the specified address and return the short value to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic.

WARP_DOUBLERETRIEVEEXPORT TCS_ProviderSupportRetrieveDouble (double *p_SupportDoubleVal)

Description: When the warp command requires the Provider to fetch a double value, a Visual Basic program can call TCS_ProviderSupportRetrieveDouble passing the desired address of the double value. This address was provided in the message sent by WARP.dll to the Provider. This warp support routine will fetch the double value from the specified address and return the double value to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic.

WARP_DOUBLERETRIEVEEXPORT TCS_ProviderSupportRetrieveCoord (WARP_Coords *p_SupportCoordsVal, short iVal)

Description: When the warp command requires the Provider to fetch a coordinate pair, a Visual Basic program can call TCS_ProviderSupportRetrieveCoord passing the desired address of the coordinate structure and an index (iVal) indicating which member (Xval or Yval) the Provider is attempting to fetch. The coordinate structure address was provided in the message sent by WARP.dll to the Provider. This warp support routine will fetch the double value from the selected Xval (if the index is zero) or the Yval member (if the index is non-zero) of the coordinate structure. This support routine will return the selected double value to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic. In order for the Provider to fetch both the Xval and Yval, it is necessary for the Provider to call this support routine twice, once with an index of zero and once with a non-zero index.

Return Support Functions

Return support functions are provided within WARP.dll that can be called by a Visual Basic program passing the address and desired value of data to the WARP.dll. The support routine takes the provided data and stores it in the specified memory location and returns the WARP_ACK status value. Specific support routines are provided to return a short, double, and coordinate structure data types.

WARP_EXPORT TCS_ProviderSupportReturnShort (short *p_SupportShortVal, short rVal)

Description: When the warp command requires the Provider to set (store) a short value, a Visual Basic program can call TCS_ProviderSupportReturnShort passing the memory address and desired short value. This address was provided in the message sent by WARP.dll to the Provider. This warp support routine will store the provided short value into the specified address and return the WARP_ACK status to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic.

WARP_EXPORT TCS_ProviderSupportReturnDouble (double *p_SupportDoubleVal, double dVal)

Description: When the warp command requires the Provider to set (store) a double value, a Visual Basic program can call TCS_ProviderSupportReturnDouble passing the memory address and desired double value. This address was provided in the message sent by WARP.dll to the Provider. This warp support routine will store the provided double value into the specified address and return the WARP_ACK status to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic.

WARP_EXPORT TCS_ProviderSupportReturnCoords (WARP_Coords *p_SupportCoordVal, double xVal, double yVal)

Description: When the warp command requires the Provider to set (store) a coordinate value, a Visual Basic program can call TCS_ProviderSupportReturnCoords passing the memory address and desired Xval and Yval values. This address was provided in the message sent by WARP.dll to the Provider. This warp support routine will store the provided values in the data elements of the specified coordinate data structure and the WARP_ACK status to the Visual Basic program using the __stdcall calling and parameter passing convention expected by Visual Basic.