Synchronized Time Stamp Support in EPICS

(original by J.Winans, modifications by T.Korhonen)

1. Introduction

Every time an EPICS record is processed, it gets a time stamp to record the time of processing. This timestamp is used by client programs like the channel archiver to align the retrieved values in time.

The task of maintaining time stamps over all IOC:s on a network is handled by the timestamp driver in iocCore (drvTS). The purpose of this document is to explain how the synchronization (as it is currently implemented) works. Some of the features are not obvious and this document tries to clarify them.

2. Overview

Each IOC is responsible of maintaining its own local time. Time across IOC's is synchronized using UDP transactions with a master/slave relationship. A **master timing IOC** is responsible for knowing the actual time, a **slave IOC** uses the master to verify it's own time stamps.

(There should be only one master timing IOC in the same broadcast subnet.)

Time can be maintained in two ways: using an **event system (Synchronous time)** or using **IOC tick counters (Soft Time)**. An IOC can be configured to run in one of four modes:

synchronous-master (with event system)
soft-master
synchronous-slave (with event system)
soft-slave

Each mode maintains time differently. When the time stamp support code initializes, it determines the mode in which it will operate based on configuration parameters and available event system function information.

2.1. Synchronous Time

Synchronous timing is available when **event system hardware** is present. For a synchronous slave, an event receiver is required, for a master an event generator and an event receiver are required.

The IOCs using synchronous time from the same master will have exactly aligned time stamps. To achieve this, the event receiver has a tick counter which is incremented using either a "tick" event or internal cycle counters. To synchronize all receivers and to prevent the counters from flowing over, there is a "reset tick counter" (or "sync") event that resets the counter to zero and posts the event to the time stamp support software. The time stamp support software knows the current time by looking at the last "tick reset event" time and querying

the event system with ErGetTicks() for the number of ticks which have elapsed since the last reset.

Since the event handler function gets called with the "ticks since last reset", an event time is the last "tick reset event" time plus the ticks count in the call.

Note: the time stamp software expects to find an **event receiver card** with the **card number 0** before it configures itself in synchronous time mode.

2.1.1. Master

A synchronous master is responsible for providing the "tick" event (if necessary) and the "reset tick counter" event. The master is determined by the first parameter to TSconfigure() and the presence of event hardware. **A master must have an event generator and an event receiver**. Since the event system is configured from EPICS database records, the database must have records in it to initialize the event system.

When the time stamp support software's event handler on a master timing IOC receives a "reset tick counter" event, a time stamp message is broadcast out on the slave_port (configured in TSconfigure()). The master timing IOC is also listening on the master_port (also configured in TSconfigure()) for incoming requests for time stamp information.

At boot time, a master will set the vxWorks clock from the Unix boot server time. The time is retrieved from the boot server using NTP or the time protocol, or the server pointed to by the EPICS environment variable EPICS_TS_NTP_INET using NTP. (For direct time: the GPS module takes an unknown period of time to sync to the correct time, so it can not be used until it's time is valid.)

The event system is not up and running until record support initialization is complete, therefore the event system time stamps are not ticking until the event system initializes. At the time of the first sync, the event system is known to be up, and the "reset tick counter" or "sync" event is set to the current time (vxWorks clock).

2.1.2. Slave

A synchronous slave is determined by the first parameter to TSconfigure() and the presence of event hardware. This type of slave must have an event receiver. The EPICS database must be configured to initialize the event system.

When an IOC is configured to be a synchronous slave, it broadcasts a request for a master on the master_port port. If a master replies, then all current time, sync rate, and clock are extracted from the master's response (the sync rate and clock rate from the TSconfigure() are overwritten to match the master's configuration). If a master is not found, the slave IOC goes into a polling mode to try to find a master every two minutes. While a slave has no master, the IOC's clock is initialized from the Unix boot server and TSgetTimeStamp() returns the vxWorks time clock value.

2.1.3. Event System Requirements

The time stamp software has some requirements in order to provide event system synchronized time.

-Event system hardware has (finite) counters for counting pulses or ticks delivered to it from some master tick generator. The software requires access to these counters at any time.
-there must be a possibility to (synchronously) reset the counters to zero before the counters overflow with a signal or event. The software requires a notification when the "counter reset" arrives.
-the event system latches the tick counter when an event is received. The events are actually stored into a FIFO which the interrupt service routine reads (and clears).

The combination of hardware tick counters, timestamp latching and reset event notification gives the software the ability to maintain time. If the event system supports other events, it is required that the software be notified of the occurrence of each event. The event time will be recorded in a table which holds one time stamp value for each possible event so that a user can inquire when the event occurred.

2.2. Soft Time

If no event system is present, the time stamp software can operate in a software timing mode. Soft time IOCs use the 60 hertz clock available from vxWorks to maintain a time stamp. The IOC will increment the time stamp (usually) at a 60 hertz rate. All soft timing IOC's will not have the same time stamps. No events are available, so the software always retrieves the current time.

2.2.1. Master

A soft master is determined by the first parameter of TSconfigure() and the absence of event hardware. Upon boot, the master soft timing IOC retrieves the current time from the Unix boot server. The IOC sets up a soft time-stamp counter using a one tick watch dog and sets the vxWorks clock. From this point on, the master runs using the soft time-stamp counter. The master listens on the master_port port for requests for the master's current time.

2.2.2. Slave

A soft slave is determined by the first parameter of TSconfigure() and the absence of the event hardware. The basic operation of a soft slave is to synchronize the time stamp with a master when possible or to the Unix boot server if no master is available. Upon initialization the time stamp support code determines if a master is present on the network, and if NTP support is available from the Unix boot server or the server pointed to by the EPICS environment variable EPICS_TS_NTP_INET. The slave will request time stamps from a master or unix server at sync_rate_in_seconds intervals from TSconfigure(). If the time stamp on the slave is found to be off, the clock will be incrementally

adjusted so that by the next sync, the clock will be corrected to match the master's time stamp. A soft slave will automatically switch between a master and the unix boot server depending on if the master is available or not. In order to sync with the unix boot server, NTP must be available there for query.

3. Design Specifications Summary

(proposed enhancements are written in *italic*)

3.1. Event System Synchronized Time

All IOC:s will have identical event times. In addition, all IOC:s will maintain the same current time.

At minimum two events must be supported:

-a "**tick**" event to increment the timestamp counter. Or, the counter can be incremented without events, using a synchronized internal clock.

-a "**reset hardware tick counter**" event. This will be used to update the time stamp representing the time when the sync last occurred. All other timestamps and the local "current" time will be calculated from that point of time.

An optional event can be used as a "heartbeat" event. This event can be used to signal errors.

3.2. Soft Time

All IOC:s of this type will maintain time stamps which are within two clocks ticks or 1/30th of a second of a master. The master may be a designated IOC or the Unix boot server. A Unix **boot server master** or **the server pointed to by the EPICS environment variable EPICS_TS_NTP_INET** must have NTP available for polling.

3.3. Role of a Master Timing IOC

A master timing IOC has different responsibilities depending on if it is an event master or a soft master.

An **event master** will be an event generator (create "tick" and "counter reset" events). When the event master detects a "counter reset", it broadcasts the time the event occurred to all event slaves on the network. The slaves can use the time stamp data to verify their own time.

A **soft timing master** does not have any events, so no broadcasting is done.

The time must always be monotonically increasing, with no jumps backward in time.

Both soft and event masters have the ability to be queried at any time by slaves for the current time.

In addition, a master ioc is responsible for keeping its local time (and thus also the time of all its slave iocs) synchronized to the NTP time. Although the synchronization is less critical than that between the ioc's, large drifts cause confusion with external client programs and operations.

This synchronization will be implemented similarly to the synchronization in NTP (v.4) protocol, with a PLL/FLL filter to lock to the NTP server(s).

When a connection to the NTP server is broken, the system runs on local clock without corrections until the connection is resumed. After resumption, the IOC clock will not be stepped but slewed to match the NTP time.

In case of hardware failure that makes the master incapable of serving synchronized time, it should revert to a soft master mode and command the slave timing ioc's also to switch to soft clock.

3.4. Role of a Slave Timing IOC

A slave timing IOC also has different responsibilities depending on if it is an event slave or a soft slave.

An **event slave** contains hardware to manage "tick" and "counter reset" events. The time stamp support software uses this information to maintain time. An event slave will listen for broadcasts from the event master timing IOC and use the information to verify it's time.

In a case of a hardware failure, the event slave will revert to soft timing operation and raise an alarm state. When the event time resumes, at next synchronization the time is stepped to match the master time.

A **soft slave** periodically queries a master timing IOC to get the current time. It uses the received time to synchronize its own local time to the master time, by slowly slewing its time.

A possibility to configure the IOCs always to connect to a NTP server instead of trying to find a master will be added. A possibility for a master to signal that it is out of synchronization will be added. This will be notified by the slave iocs and they switch to NTP mode instead.

General

A possibility to monitor the timestamp software status and for a limited control over channel access (i.e., a record support) will be added.

API Guide

Theory of operation:

EPICS provides timestamps to the channels through functions implemented in drvTS.c. The timestamps can be either hardware-assisted or soft stamps. The code keeps a table of time stamps for a number of events (typically 256.) A time stamp in the low level consists of a pair (sec, nanosec), giving the elapsed time in seconds and nanoseconds after the EPICS 'epoch' of 1. January, 1990. The timestamps for the events are kept in an array (event_table[256]). A few entries in the array have special importance:
`event_table[0]` holds the current vxWorks clock time. This is the time stamp given to a channel when it is processed, if a timestamp event (TSE) has not been specified.
`event_table[sync_event_number]` is the time when a global synchronization of timestamps last occurred.

	SEC	NSEC
[0]	6543001	7891234
[1]	6542905	3450987
.		
.		
.		
.		
.	6542986	2346802
[sync_evt]		
.		
.		
.		
.		
.		
.	6542997	9124680

Figure 1. Timestamp table.

In case of soft timing, the records always get the current time, which is held in `event_table[0].` The current time is updated using the vxWorks timer, so the resolution is the clock rate (60 Hz by default.)

With the synchronized timing, every time an event occurs, an handler is called (through an interrupt). The handler reads the tick counters of the hardware and updates the event_table. In case of a synchronization event, the broadcast server is scheduled to send out a sync broadcast message.

Important data structures (objects):

TSdata:

global structure to hold configuration and status data.

```
struct TSinfoStruct {
        TSstate state;                  /*master IOC alive or dead */
        TStime_type type;                       /*configured type: sync/async,master/slave*/
        TStime_protocol async_type;     /*async protocol type:ntp,time,private,none */
        int ts_sync_valid;              /*validity of timestamp: 0-not valid */
        struct timespec *event_table;   /* pointer to the timestamp table */
        unsigned long sync_rate;        /* master send sync at this rate */
```

```
        unsigned long clock_hz;           /* master clock is at this frequency */
        unsigned long clock_conv;         /* conversion factor for tick_rate->ns */
        unsigned long time_out;           /* udp packet time-out in milliseconds */
        int master_timing_IOC;            /* 1=master, 0=slave */
        int master_port;                  /* port that master listens on */
        int slave_port;                   /* port that slave listens on */
        int total_events;                       /* this is the total event in the event
system*/
        int sync_event;                   /* this is the sync event number */
        int has_event_system;             /* 1=has event system, 0=no event system */
        int has_direct_time;              /* 1=has direct time, 0=no direct time */
        int UserRequestedType;            /* let user force the setting of type */
        SEM_ID sync_occurred;             /* semaphore for sync master&client tasks*/
        struct sockaddr hunt;             /* broadcast address info */
        struct sockaddr master;           /* socket info for contacting master */
};
```

Packet structure of synchronization messages:

TSstampTransStruct is used when transferring synchronization timestamps between master and slaves. Upon request from a slave ioc (TSasyncClient, TSsyncClient), the timestamp server task on the master timing ioc fills this structure and sends it to the requester.

```
struct TSstampTransStruct {
        unsigned long magic;              /* identifier */
        TStype type;                      /* transaction type */
        struct timespec master_time;      /* master time stamp - last sync time */
        struct timespec current_time;     /* master current time stamp 1990 epoch */
        struct timespec unix_time;        /* time using 1900 epoch */
        unsigned long sync_rate;          /* master sends sync at this rate */
        unsigned long clock_hz;           /* master clock this frequency (tick rate) */
};
```

NTP packet structure:

```
struct TS_ntp {
        /* unsigned int info; */
        unsigned char info[4];
        unsigned int root_delay;
        unsigned int root_disp;
        unsigned int reference_id;
        struct timespec reference_ts;
        struct timespec originate_ts;
        struct timespec receive_ts;
        struct timespec transmit_ts;
        /* char authenticator[96]; (optional) */
};
```

Depending on the type of the ioc (soft/hard master/slave), there are a few **tasks** that are started during iocInit to handle the synchronization:

TSsyncServer()

Broadcasts the synchronization time stamp (i.e., event_table[sync_event_number]) every time a sync event is received by the event system. The task waits on a semaphore (sync_occurred, given by TSeventHandler when an synchronizing event is received) and when the the semaphore is given, takes the synchronization time stamp and broadcasts it. For the first synchronization the time stamp is retrieved from vxWorks clock.
Started with task name: `ts_syncS`
Priority: 70 (TS_SYNC_SERVER_PRI)
Port: slave communications port (default: 18322, TS_SLAVE_PORT)

TSsyncClient()

Before starting, checks that the master timing IOC is alive. After starting, waits for the sync time broadcasts from master (TSsyncServer) with select (blocking wait). When a sync timestamp is received, compare it with the local time stamp. If the

timestamps differ, send a log message and set the local time to the time received from master.
Started with task name: `ts_syncC`
Priority: 70 (TS_SYNC_CLIENT_PRI)

TSasyncClient()

Keeps the IOC time in sync with a NTP server. First, opens one socket to contact a NTP server and one broadcast socket to look for master IOC. If a master IOC is found, a socket to contact it is opened. Operation:
-if master IOC is alive, request a synchronization timestamp from the master every *sync_rate* seconds. Uses the TSsyncTheTime to make the synchronization locally.
-if no master available, try to synchronize with the NTP server. Broadcast to find a master every TS_SECS_ASYNC_TRY_MASTER (=5 min.)
TSsyncTheTime is used to adjust the time (see the corresponding document for an explanation how it works.)
Note: in any case the master is looked for. That means: if a timing master (soft or synchronous) is started on the same subnet, within 5 minutes all slaves are synchronized to it. There is no means to prevent this from happening (except for starting no master timing IOC.)

Started with task name: `ts_Casync`
Priority: 70 (TS_SYNC_CLIENT_PRI)

TSstampServer()

Listens on the master port (defined in TSconfigure) for timestamp or sync requests. The last sync time, current IOC time, sync rate and clock rates are sent to the requester.
Clients use TSgetData to request the time stamps, called in TSasyncClient and from TsgetMasterTime, called from TSsetClockFromMaster, called from TSinit and TSsyncClient.

Started with task name: `ts_stamp`
Priority: 70 (TS_STAMP_SERVER_PRI)
Port: master communications port (18323, TS_MASTER_PORT)

In addition, there are "watchdog" tasks that are scheduled every vxWorks tick:
TSwdIncTime
Increment the IOC current time (event_time[0]) for a soft timing IOC.
TSwdIncTimeSync
Increment the IOC current time (event_time[0]) for a hardware-timed IOC.

These "tasks" are not like traditional tasks, but rather routines that are called every clock tick using the vxWorks watchdog facility. This implies that these routines are called at interrupt level.

List of functions provided/used in drvTS.c

static long **TSgetUnixTime**(struct timespec*);

Ask the NTP server or if not available, the boot server for the time using the NTP protocol.

static long **TSgetMasterTime**(struct timespec*);

Query the master timing IOC for it's current time.

Routines for managing sockets

static long **TSgetBroadcastAddr**(int soc, struct sockaddr*);
Determine the broadcast address, this is directly from the Sun Network Programmer's guide.

static int **TSgetBroadcastSocket**(int port, struct sockaddr_in* sin)
Return a broadcast socket for a port, return a sockaddr also.

static int **TSgetSocket**(int port, struct sockaddr_in* sin)
Get a UDP socket for a port, return a sockaddr for it.

static long **TSgetData**(char* buf, int buf_size, int soc, struct sockaddr* to_sin, struct sockaddr* from_sin, struct timespec* round_trip)
Attempt to get the timestamp data from a socket after sending a request to it. Optionally return round trip time and the sockaddr of the respondent.

static void **TSaddStamp**(struct timespec* result, struct timespec* op1, struct timespec* op2);

static long **TSstartSyncServer**();

static long **TSstartSyncClient**();

static long **TSstartAsyncClient**();

static long **TSstartStampServer**();

static long **TSuserGetJunk**(int event_number,struct timespec* sp);

Event system and clock function hooks

static long (***TSregisterEventHandler**)(int Card, void(*func)());
This is used to register an event handler function to be called upon receiving an event.

static long (***TSregisterErrorHandler**)(int Card, void(*func)());
This is used to register an error handler function to be called when the timing card posts an error.

static long (***TSgetTicks**)(int Card, unsigned long *Ticks);
A function to read the tick counter of a hardware timing card.

static long (***TShaveReceiver**)(int Card);
Used to test the existence of a timing receiver card.

static long (***TSforceSync**)(int Card);

static long (***TSgetTime**)(struct timespec*);

static long (***TSsyncEvent**)();
The number of the synchronization event (counter reset event).
static long (***TSdirectTime**)();
Flag for availability of direct time.
static long (***TSdriverInit**)();

static long (***TSuserGet**)(int event_number,struct timespec* sp);

Configuration & status query functions

void **TSconfigure**(int master, int sync_rate_sec, int clock_rate_hz,
int master_port, int slave_port, unsigned long time_out, int type)

This is the configuration routine to be called from the vxWorks startup script before iocInit. It's job is to set operating parameters for the time stamp support code.
If type = 0, then try to configure using event system, if type = 1, then permanently inhibit use of the event system.

long **TSreport**();
Report information about the current configuration and state of the time stamp support software. Callable from vxWorks shell.

long **TSdriverInitError**()

unsigned long **TSepochNtpToUnix**(struct timespec* ts)

unsigned long **TSfractionToNano**(unsigned long fraction)

unsigned long **TSepochNtpToEpics**(struct timespec* ts)

unsigned long **TSepochUnixToEpics**(struct timespec* ts)
Converts an Unix epoch timestamp ([sec,ns] since 1. January, 1970) to an Epics epoch timestamp ([sec,ns] since 1. January, 1990).

unsigned long **TSepochEpicsToUnix**(struct timespec* ts)
Converts an Epics epoch timestamp to an Unix epoch timestamp.

long **TSgetTimeStamp**(int event_number,struct timespec* sp)

This routine returns the time stamp that represents the time when an event event_number occurred. Soft timing will always return the current time. Event zero will also always return the current time.

long **TSinit**(void)
Initialize the driver, determine mode. Called by iocInit.

Watch dog routines for soft time support

static void **TSstartSoftClock**()
Start the soft clock watch dog.

static void **TSwdIncTime**()
Increment the time stamp at a 60 Hz rate. For soft timing. Called at interrupt level.

static void **TSwdIncTimeSync**()
Increment the time stamp from the event receiver ticks.

Interrupt service routines

static void **TSeventHandler**(int Card,int EventNum,unsigned long Ticks)
Receive events from event system; update the timestamp table. This is registered as the event system event handler and will be called from the event receiver interrupt handler (i.e., this routine is executed in interrupt level.)

static void **TSerrorHandler**(int Card, int ErrorNum)
Receive errors from event system. This is registered as the event system error handler and will be called from the event receiver interrupt handler (i.e., this routine is executed in interrupt level.) Note: called at interrupt level!

Utilities for initially getting and setting the time

static long **TSsetClockFromUnix**()
Query the time from NTP/boot server and set the vxworks clock.

static long **TSsetClockFromMaster**()
Set the vxworks clock using the time from the master timing IOC.

static long **TSsyncTheTime**(struct timespec* cts, struct timespec* ts)
Given the current time (cts), and a compare time stamp (ts), correct the current time clock to match ts.

long **TScurrentTimeStamp**(struct timespec* sp)
Get the current time from time stamp support software.

long **TSaccurateTimeStamp**(struct timespec* sp)
Get the current time from time stamp support software.

static long **TSgetCurrentTime**(struct timespec* ts)
Get the current time from vxWorks time clock

static long **TSforceSoftSync**(int Card)
Routine for causing sync to occur (not a hardware one.)

static void **TSaddStamp**(struct timespec* result, struct timespec* op1, struct

timespec* op2)
TSaddStamp - Add time stamp op1 to time stamp op2 giving time stamp result

static long **TScalcDiff**(struct timespec* a, struct timespec* b, struct timespec* diff)
Calculate the difference between to time stamps. The difference between arguments 'a' and 'b' is returned in 'diff'. The return value from this routine is the direction: (1) =a>b, (-1)=a<b.

Test functions

void **TsprintRealTime**()
Print the IOC real time clock value. Call the ErGetTime() function described below (if present) and print the time stamp returned from it. Also print the current EPICS time stamp.

void **TSprintTimeStamp**(int num)
Print the EPICS time stamp for the given event_number.

void **TSprintCurrentTime**()
Print the current time using event system.Print the EPICS time stamp representing the current time.

void **TSprintUnixTime**()
Send a time stamp query transaction to the boot server or NTP server and print the returned time stamp.

void **TSprintMasterTime**()
Send a time stamp query transaction to the master time server and print the returned time stamp.

long **TSgetFirstOfYearVx**(struct timespec* ts)
/* gross and horrid example */

Configuration of IOCs for time stamping support

To retrieve the time stamp which represents the time that an event occurred, use **TSgetTimeStamp()**. **TScurrentTimeStamp()** can be used to retrieve the time stamp which represents the time of day. The function **TSgetFirstOfYear()** attempts to give the caller a time stamp representing January 1 of the current year relative to the vxWorks epoch 1970.

An EPICS environment variable named EPICS_TS_NTP_INET exists which can be set to point to an NTP server. The default NTP server is the IOC boot server.

Debugging Information

A global variable exists named **TSdriverDebug**. Setting this variable to a positive value in the vxWorks start up script will inform the time stamp driver to print information about what it is doing. The greater the value, the more information the driver will print. The number can be set and adjusted to any value any time while the IOC is running.

Driver Configuration

The synchronous time stamp support software is configured by calling **TSconfigure ()** from the "startup.script". The parameters to this routine are:

- **master_indicator**: 1=master timing IOC, 0=slave timing IOC, default is slave.
- **sync_rate_seconds**: The clock sync rate in seconds. This rate tells how often the synchronous time stamp support software will confirm that an IOC clock is synchronized. The default is 10 seconds.
- **clock_rate_hz**: The frequency in hertz of the clock, the default is 1000Hz for the event system. The value will be set to the IOC's internal clock rate when soft timing is used.
- **master_port**: The UDP port which a master timing IOC will use to receive time stamp requests. The default is 18233.
- **slave_port**: The UDP port which a slave will use to receive time stamp information broadcasts from a master.
- **time_out**: UDP information request time out in milliseconds, if zero is entered here, the default will be used which is 250ms.
- **type**: 0=normal operation, 1=force soft timing type.

This routine must be run **before iocInit**(). The synchronous time stamp support software is initialized as part of iocInit.

Running **TSreport()** after iocInit() will produce a report which shows the current state of the driver.

Event System interface

The event system interface consists of seven functions which can be provided by an event system. The synchronous time stamp support software uses a card number of zero on all functions that require a card number. The functions are as follows:

- long ErHaveReceiver(int event_card_number)
- long ErGetTicks(int event_card_number, unsigned long* ticks)
- long ErRegisterEventHandler(int event_card_num,EVENT_FUNC event_func)
- long ErRegisterErrorHandler(int event_card_num,ERROR_FUNC error_func)
- long ErForceSync(int event_card_number)
- long ErGetTime(struct timespec* time_stamp)
- long ErSyncEvent()

- long ErDirectTime()
- long ErDriverInit()

The definitions are as follows:

- **ErHaveReceiver()**: Returns -1 if no event (timing) hardware present, else returns the number of supported events.
- **ErGetTicks()**: Returns the number of ticks since the last hardware tick counter reset.
- **ErRegisterEventHandler()**: Informs the event system of a function to call when an event occurs, the format of the function will be defined below.
- **ErRegisterErrorHandler()**: Informs the event system of a function to call when an error occurs, the format of the function will be defined below.
- **ErForceSync()**: This function will force an event generator to generate a tick reset event and send it.
- **ErGetTime()**: This function returns the actual time. The intention here is that this function will retrieve the actual time from GPS system or equivalent and return is in time stamp format.
- **ErSyncEvent()**: Return the event number for the tick reset event.
- **ErDirectTime()**: Return 0 for normal operation, return value not = 0 for systems that has direct time access, such as a GPS.
- **ErDriverInit()**: The time stamp driver initialization function will call this user supplied function before it returns and after it sets up the vxWorks clock and attempts to set up the TIMEZONE variable. This can be useful to initialize a system such as a GPS (the year can be determined using the vxWorks ansiTime library).

All of these routines are checked to exist when the time stamp support code initializes. All have some kind of default routine provided if they are not found, most of which just return an error condition. The functions which the time stamp support software registers (event and error) have the following format:

- TSeventHandler(int Card,int EventNum,unsigned long Ticks)
- TSerrorHandler(int Card, int ErrorNum)

Here the Card is the event system board of interest (always zero), the EventNum is the event that occurred, and the Ticks is the number of ticks since the last board tick counter reset.

In addition to the above functions, a global variable exists on the IOC which can be used to indicate that direct time is available. Setting the variable **TSdirectTimeVar** to a nonzero value has the same effect as providing the ErDirectTime() function.

Creating Direct Time Support

Most of the above interface functions apply only when special event hardware is present. A much simpler configuration is when a GPS is present and time stamps are distributed using IRIG-B to all or most of the IOCs. The easiest way to implement this scenario is to define ErDirectTime() to return the value one and define ErGetTime(). The job of ErGetTime() will be to actually be to generate and return the time stamp representing the current time (from the EPICS epoch). At

system initialization, the actual time is retrieved from a Unix server through NTP or the Unix time protocol, so that year will be valid in the vxWorks time clock. At the GPS driver initialization, the vxWorks function clock_gettime() can be used to calculate the year. Record support will internally eventually call ErGetTime() each time the record is processed. The GPS driver should be included in the *.dbd file, so it initialized in the proper order.

Record Support

Record support has the ability to tie the record processing time to an event. This means that a user can specify that processing of a record is due to an event (from the event system). When the record gets processed, the time in the TIME field of the record will be the time when the event occurred. In order to support the event times from record support, there are two fields that are common to all records: Time Stamp Event (TSE), and Time Stamp Event Link (TSEL). The TSE field is the actual event time the user is interested in. The TSEL field will be a link used to populate the TSE.

To facilitate the use of the time stamp support software, a new record support function will be added:

- recGblGetTimeStamp((dbCommon*)prec).

This routine uses TSgetTimeStamp(prec->tse, &prec->time) to set the processing time of the record. If the TSE field is zero (the default), then TSgetTimeStamp() will report the current time. It is important to remember that if a TSE field is set, then the processing time (in field TIME) will always reflect the last time the event occurred.

Definitions

Time Stamp: Two long words representing the time in seconds/nanoseconds past an epoch. The first long word represents the seconds past the epoch, the second long word represents the nanoseconds within the second. The EPICS epoch is currently January 1, 1990. A commonly used Unix epoch in January 1,1900 or in vxWorks case, January 1,1970.
Event System: A hardware based subsystem for delivering events to all IOC:s on a network. Typical events are the "tick" event and the "reset counter" event.
Event System Synchronized Time: Time stamps maintained using an event system. The clock pulses which increment the time stamps are provided by a single source, all IOCs increment time stamps using the single source pulse.
Soft Time: Each IOC increments the time stamps using it's own internal clock.
Master Timing IOC: The IOC which knows the actual time and is the source of the actual time to all those who inquire.
Slave IOC: An IOC which relies on a master IOC to provide the actual time. This IOC will keep it's time stamp in sync with a master.
Soft Slave: An IOC which uses Soft Time and synchronizes it's time stamp with a master.
Event Slave: An IOC which uses Event System Synchronized Time to maintain it's time stamp. This type of IOC uses the master to verify that it's time stamp is correct.
Soft Master: A Master Timing IOC that maintains it's time stamp using a

private clock.
Event Master: An IOC which uses Event System Synchronized Time and is a Master Timing IOC. In addition, this IOC is the source of the clock pulses.