OpenVMS Technical Journal V5

Introduction to the Performance Data Collector for OpenVMS (TDC)

Introduction to the Performance Data Collector for OpenVMS (TDC)

Lee Clark, Software Engineer, System Management Tools Group, OpenVMS Engineering

Overview

The Performance Data Collector – **TDC** – Version 2.1 represents an evolution of the TDC V1 project completed during 2001-2002 by OpenVMS Engineering at the request of an independent provider of system management solutions. The approach used for that project was extended and generalized to provide not only a significantly expanded set of system performance metrics but also an open and extensible framework for building performance management and analysis applications for the OpenVMS platform.

TDC provides an Application Programming Interface (API) that can be used to manage collection and processing of approximately1000 performance-related metrics organized into 21 functional categories, or data record types. Among these are storage utilization and I/O; cluster communications; network performance; lock manager performance; process metrics; memory, CPU, server, file system, and cache utilization and performance; paging performance; and SYSGEN parameters. The provided data will not be discussed in this article (see "Software Developers Kit" later in this article).

With its ability to load software modules at runtime, the API supports external development and deployment of additional collection and processing capabilities. The API also supports both file-oriented and "live" processing of collected data.

TDC Version 2.1 is installed as a required System Integrated Product (SIP) with OpenVMS Version 8.2 on Alpha and Integrity Server systems. Software for Alpha systems running OpenVMS Version 7.3-2 is available via download from the web. (The URL is provided at the end of this article.) VAX systems are not supported.

This article begins with an overview of the TDC architecture and programming environment and concludes with a brief discussion of the Performance Data Collector from a system management perspective. Goals of this article are to provide enough information to help you determine whether the software might be useful in your application and to help you determine the level of difficulty you might encounter in integrating the software with your application. This article is not, however, intended as an exhaustive tutorial.

The TDC Software Architecture

The architecture embodies the following key ideas:

- The TDC ***Engine*** manages operations, under the control of a ***Client Application***.
- ***Processor Modules*** produce or utilize data, at the direction of the Engine.
- For any operation, whether formally a data-collection operation or a data-extraction operation, the TDC Engine rigidly enforces a temporal separation between the production of data records and the utilization of those records at each step of the operation.

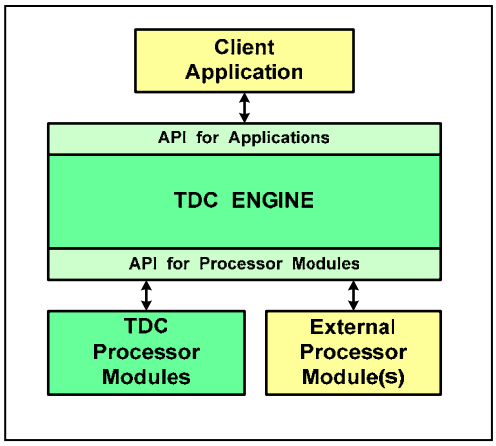

Figure 1 -- TDC Architectural Overview

Figure 1 illustrates the principal components of the architecture and their relationships. (Green boxes in Figure 1 and Figure 2 represent the TDC software; yellow boxes represent software that can be integrated with TDC.) The Client Application depicted in provides a user interface to the data-collection system. The Client might be the control application, TDC$CP, installed with TDC or it might be an independently-developed application—possibly even a server application. Any Client uses the TDC API to control the TDC Engine.

Based on information provided by the Client Application through the API, the Engine (installed as shareable image TDC$APISHR) locates and loads the Processor Modules required to perform the current operation. Processor Modules might be loaded from the library that is provided with TDC (shareable image TDC$LIBSHR), or they might be loaded from one or more externally-developed shareable images specified by the Client Application. From the Engine's perspective, the *only* difference between Processor Modules provided with TDC and "External" Processor Modules is that the Engine knows which shareable image contain those Processor Modules provided with TDC. Once the appropriate Processor Modules have been loaded, the Engine makes no further distinction between TDC-supplied and External Processor Modules.

Processor Modules interact with the TDC Engine through a pointer-based API that overlaps extensively with the name-based API available to Client Applications. Processor Modules respond to messages directed at them by the Engine by performing the specified operation and returning a status to the Engine.

As part of the loading "handshake" between the Engine and a Processor Module, the Processor Module specifies the types of operations it can perform (***capabilities***) and at which points during the current operation it should be called (***synchronization***). Among the possible capabilities that the architecture defines for Processor Modules, the following are the most important in understanding the Engine's organization of the system's runtime behavior:

- A ***Timer*** Processor Module controls the pacing of the current operation. When called by the TDC Engine, a Timer suspends the current operation until it is time to generate a new set of data records. The Timer's suspension of activity might be time-based, it might be based on user input, or it might even be based on interrupts from another application. Although a time-based Timer is provided with TDC, it can be overridden by an externally-provided Timer. Exactly one Timer must participate in data-collection operations. (The Client Application can provide the "timing" function in some modes of operation.) A Timer is optional when extracting data from a file.
- ***Producer*** Processor Modules provide data records to the system. The data records might be collected from the operating system through system services or other means, they might be based on information collected from other applications, or they might be derived from data records provided during the current operation by other Producer Processor Modules. The sole requirement for a Producer is that it provide — through the API — one or more instances of exactly one type of data record whenever the Engine instructs it to do so. The data records

provided by all Producers participating in the current operation are stored in a data aggregate, or *snapshot structure*, by the Engine.

- An ***Extractor*** Processor Module reads data records from a file and uses the API to populate a snapshot structure with them. From the Engine's perspective, an Extractor is nothing more than a "super" Producer: it produces data records of (potentially) several different types when the Engine instructs it to do so. An Extractor is provided with TDC and, at this time, there is no provision to override it with an externally-furnished Extractor.
- ***Consumer*** Processor Modules utilize the data records provided by Producers and made available to them through the API. The Engine maintains two complete snapshot structures: a "current" snapshot containing the most recently-produced set of data records, and a "previous" snapshot containing the set of data records produced prior to that. A Consumer might concern itself either with only the most recent data, or with changes in values from one snapshot to the next. The TDC architecture does not restrict Consumers with respect to how the data is processed. For example, different consumers might write the provided data in raw form into a file, analyze it to create a report, or transmit it to another application.

After all Processor Modules have been successfully loaded, the Engine examines the load-handshake information each Processor Module has provided to determine when, and in which order, the Processor Modules should be called as the current operation proceeds.

The API supports two standard operations: data collections, in which data is collected (by Producers) for processing (by Consumers), and data-extractions, in which data in a file is extracted (by an Extractor) and processed (by Consumers). Although the two operations are distinguished at the API level, the Engine itself operates in much the same manner for the two, as depicted in Figure 2.

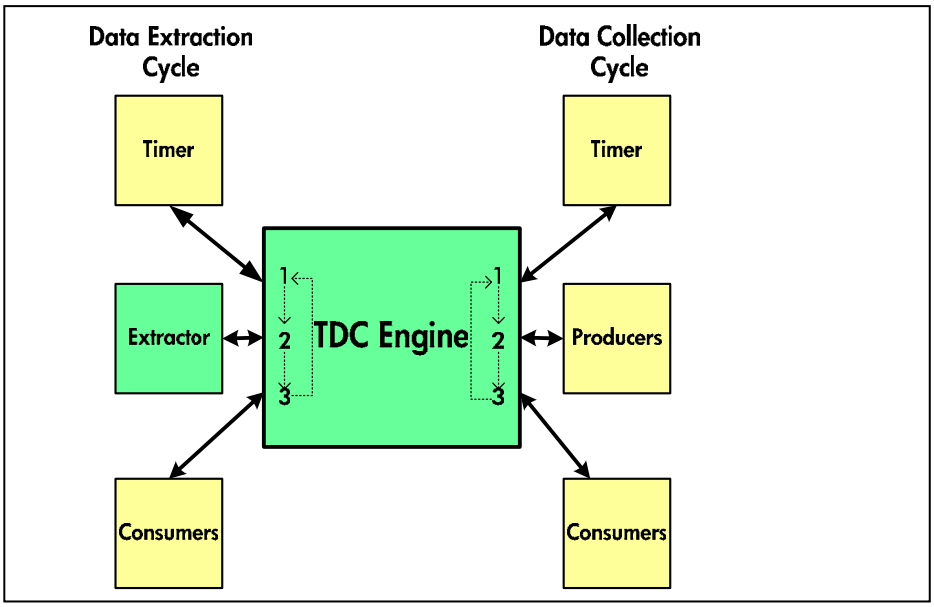

Figure 2 -- Processor Module Calls by the TDC Engine

Figure 2 illustrates how the TDC Engine handles any operation as a repeating cycle of:

1. Calling the Timer (may be absent during extractions) and waiting for it to return control to the Engine.
2. Calling Processor Modules to provide data records (Producers during a collection operation; the Extractor during an extraction operation) to the Engine, which aggregates them into snapshot structures.
3. Calling Consumers to process the data most recently provided by the Producers or Extractor and that has been aggregated into a snapshot structure by the Engine.

A collection operation ends when:

- The specified—usually by the Client Application—number of cycles ("intervals") has been reached, or

- The Timer indicates that the specified—usually by the Client Application—end-time has been reached, or
- The operation is interrupted by the user (Ctrl/Z, Ctrl/C, Ctrl/Y) or by an external source (for example, the TDC control application).

And an extraction operation ends when:

- The specified—usually by the Client Application—number of intervals has been processed, or
- The last interval before a specified—usually by the Client Application—end-time has been processed, or
- All data in the file has been processed, or
- The operation is interrupted by the user (Ctrl/Z, Ctrl/C, Ctrl/Y).

In Figure 2, it is important to note that, in general, any number of Producers can participate in a collection operation, and any number of Consumers can participate in either a collection or an extraction operation. Furthermore, because Consumers only process data residing in snapshot structures provided to them by the Engine, any Consumer Processor Module should be equally at home, whether participating in a collection or in an extraction operation.

Figure 2 contains no indication of how data files are created during a collection operation and read during an extraction operation. Data files are not a part of the overall TDC "architecture." A data file must be created by a Consumer Processor Module, and read by an Extractor Processor Module. The format and content of the data file is then controlled by those Processor Modules rather than by the TDC Engine.

You can develop new software that uses TDC, or integrate TDC into an existing application in either or both of the following ways:

- Write a Client Application to drive the TDC Engine and, by extension, the Processor Modules provided with TDC to collect and manipulate data.
- Develop new Processor Modules to supplement or replace those supplied with TDC. (Note that the TDC Engine and the Processor Modules shipped with TDC are independent entities. In particular, the Engine has no special relationship with, nor dependency upon, any of the Processor Modules shipped with TDC. It could, therefore, be used for a completely different purpose than collecting and manipulating performance data.)

The TDC Programming Environment

A discussion of the TDC programming environment appropriately begins with a description of the environment and the types of software development environments for which it is suited. Key points in this description are the following:

- The TDC API has been developed for use by "C" (and "C++") code. Access to the API by code written with other development languages would require either that "C" wrappers be developed for the API function calls or that the API definition be manually ported to the other language.
- None of the software provided with TDC is multi-threaded, although that need not preclude its use with multi-threaded software. Processor Modules are invoked sequentially at appropriate times, and each usually "has the stage" for the duration of its invocation.
- Processor Modules can, and several of those shipped with TDC do, perform "background" activities to assist in their operations, perhaps to poll for device status or I/O activity. Those TDC-supplied Processor Modules that do background processing use timer events and AST routines for that purpose.
- ASTs are used extensively throughout the TDC software, both for scheduling purposes and for responding to various events. Integrating software that disables ASTs for significant periods of time into the TDC environment is likely to have a deleterious effect on the performance of the TDC software.

- The API is operation-oriented rather than object-oriented. Two operational modes are supported. In one mode, operational parameters are set (by the Client Application), and the Engine then takes control until the operation completes. A step-by-step operational mode, under direct control of a Client Application, is also supported. In general, concurrent operations by a single instance of the software are not supported. Sequential operations are fully supported and are independent of one another because the Engine destroys all context information, including (conceptually) unloading all Processor Modules at the conclusion of each operation.
- Software using the API often has direct access to data structures used by the TDC software. Most operational parameters should not be changed while an operation is under way, but they can be changed freely between successive operations.
- Multiple instances of the TDC software can safely run concurrently within a single OpenVMS system. They do not share data, and, with one exception, one TDC instance will not interfere with another instance's operations. The exception is that the control application shipped with TDC Version 2.1 provides a STOP command that allows one instance of the TDC control application to halt data collection by other TDC control application instances running anywhere in the OpenVMS Cluster.
- The images that comprise the TDC software are not installed as privileged images. Data collection typically requires elevated privileges (refer to "System Management Overview" below), and the required privileges must be enabled by the user prior to running the software. The Engine will not start an operation unless all privileges required by loaded Processor Modules have been enabled (see the "private context" description under "Principal Data Structures" below).

Snapshots and Intervals

The term ***snapshot*** is used in two contexts within the TDC environment:

- The Engine periodically invokes Producer Processor Modules to contribute data records and then invokes Consumers to process those records. A sequence of Producer invocations followed by the Consumer invocations needed to process the collected data is a ***system snapshot***.
- The data aggregate containing all data records produced at a system snapshot is a ***snapshot structure***.

Where context makes the meaning of the term unambiguous, *snapshot* is used in the rest of this article.

An ***interval*** is the period of time that separates successive system snapshots. When data is collected, a "baseline" system snapshot is performed at the start of the operation, and the first interval begins with completion of that baseline snapshot. A TDC data file will, therefore, contain one more snapshot than the number of intervals during which data was collected.

Principal Data Structures

A ***global context*** (type TDC_CTX_t) is the principal vehicle for sharing information among the Client Application, the TDC Engine, and Processor Modules. In general, that context structure is created by the Client Application, which then populates it with parameters appropriate for the current operation, including type of operation (collect or extract), start and end times (for an extraction operation, these specify the time range of the snapshots of interest from the data file), number of snapshots (to collect or to extract), and interval size. There is also provision for status information to be set by the TDC Engine, for operation-progress information to be set by the Engine and by Processor Modules, and for private status and context information to be controlled by the Client Application.

Information in the global context is used by the TDC Engine in preparing to perform an operation, and parts of the global context are used by the Engine to exchange information with Processor Modules as it invokes them:

- The global context provides two pointers to snapshot structures, one for the "current" system snapshot and one for the "previous" system snapshot. Producers typically contribute data records to the current snapshot, while Consumers typically process the data in the current snapshot. The previous snapshot is maintained so that changes in data values can be noted and processed.
- Function pointers stored in the global context by the TDC Engine provide the API through which Processor Modules interact with the Engine (for example, to store or access data records) and with one another (for example, to determine the availability and status of other Processor Modules).
- The global context provides a data buffer into which Producers place data records for storage by the Engine.

An important part of the global context structure is the ***collection header*** (type TDC_CollectionHeader_t). Maintained by one of TDC's Processor Modules (TDC_COLHDR), this area provides basic information about the system configuration and about the state of the operation to be stored as a record in a data file.

Each Processor Module is assigned a ***private context*** area (type TDC_ProcessorCTX_t) used for the exchange of information exclusively between it and the Engine. The private context is used by the Processor Module to declare its capabilities and the synchronization points at which it should be invoked. It is also used to specify any user privileges that the Processor Module requires for the current operation and to specify any other Processor Modules upon which this Processor Module depends. It is through the private context that the Processor Module supplies status information to the Engine. The private context contains a provision for context data that is truly private to the Processor Module as well as for a "public" area for sharing data with other Processor Modules.

A ***processor module description*** (type TDC_ProcessorDesc_t) is created by the Engine as each Processor Module is loaded. It contains relatively static information about the Processor Module (for example, name, shareable image from which it was loaded, version identification, supported data formats).

The data records provided by an individual Producer Processor Module at a single system snapshot are stored as a ***record set*** (type TDC_Set_t). The record set consists of a header that identifies the type of records represented by the set, the times at which the record set was created and last updated, and the count of records currently in the set. The record set header also contains a pointer to an array of pointers to the actual data records that comprise the set.

Record sets can be marked as "*persistent.*" This type of marking is appropriate for relatively static data, such as system configuration information, that is not expected to change frequently during a collection operation.

All record sets contributed by Producers are aggregated into a ***snapshot structure*** (type TDC_SnapshotHeader_t) by the Engine. The snapshot structure consists of a header that contains timestamps that specify when the snapshot structure was created and when it was last updated, as well as a pointer to an array of pointers to record set headers representing the data records that comprise the snapshot.

As it creates a new snapshot structure at the start of a system snapshot, the Engine populates it with any persistent record sets found in the snapshot structure created at the pervious system snapshot. In doing so, the Engine "clones" persistent record sets. A *clone of a record* set receives its own record set header, but receives a copy of the array of data record pointers contained within the record set that has been cloned. The API provides a function to create a clone of a record set; multiple levels of cloning are supported. Note that a clone of a record set is read-only—it cannot be updated with new records.

Processor Modules

A Processor Module is passed three arguments when it is invoked—always in user mode—by the TDC Engine: a pointer to the global context for the current operation, a pointer to the private context created for it by the TDC Engine, and a pointer to the record set it is expected to populate (Producers only). Any Processor Module is likely to be invoked multiple times during any operation. A ***message***

code in the global context (member TDC_CTX_DoWhat) identifies the activity the Processor Module is expected to perform at each invocation. Those message codes include:

- **LOAD** is the first message seen by the Processor Module, at the start of any operation. It is matched by an **UNLOAD** message when the operation has completed. Upon receiving the LOAD message (see Example 2), the Processor Module populates its private context with information that specifies how it will contribute to the current operation (as read from the global context): its capabilities, synchronization points at which it should be called, required user privileges, and the names of any other Processor Modules on which it depends. The Engine uses those names to attempt to locate the specified Processor Modules and also uses them in sorting the full set of loaded Processor Modules into a valid invocation order. (At each system snapshot, a Processor Module is invoked only after invocation of those other Processor Modules on which it depends.) A Consumer also specifies the names of any output formats it supports.
- **INITIALIZE** is the second message seen by a Processor Module for any operation. Upon receiving this message, the Processor Module initializes itself as required, perhaps by allocating memory or other resources. This message is matched by an **END_OPERATION** message at the end of the current operation, at which point the Processor Module frees all resources it might be holding.
- **CONSUME_DATA** is passed once to each Consumer after data has been collected at each system snapshot. It provides the opportunity for the Consumer to process the data just collected (see Example 4).
- **WAITING** is the message passed to a Timer after a system snapshot has been fully processed; it is the signal for the Timer to suspend activity until the next system snapshot occurs.
- Other messages that might be received depend upon the capabilities and synchronization points specified by Processor Modules when they are loaded. Possible *synchronization* points include:
    - *PREBASELINE synchronization*: the Processor Module is invoked prior to collection of the baseline snapshot, with a **BASELINE** message code; this is intended for Timers so that the start of an operation can be deferred.
    - *SNAPSHOT synchronization*: the Processor Module is invoked three times at each system snapshot, receiving the following message codes in the order specified:
        1) **PRESNAPSHOT**: provides an opportunity, if appropriate, to suspend any background activity prior to data collection.
        2) **SNAPSHOT**: Producers provide data records to the Engine for storage (see Example 3). *SNAPSHOT* synchronizers are sent this message only after all such synchronizers have been sent the PRESNAPSHOT message.
        3) **POSTSNAPSHOT**: provides an opportunity, if appropriate, to resume any background activity after all data has been collected. *SNAPSHOT* synchronizers are sent this message only after all such synchronizers have been sent the SNAPSHOT message.
    - *SNAPSHOT_END synchronization*: the Processor Module is invoked after the system snapshot is complete, with a message code of **SNAPSHOT_END**; this synchronization point is used, for example, by the TDC Consumer Processor Module responsible for writing data into a data file.
    - *END_COLLECTION synchronization*: the Processor Module is invoked, with a message code of **END_COLLECTION**, when all system snapshots required for the current operation have been collected and processed; it might be used to close an open data file, or to send notification to another application that the data file is ready for processing.
- A **DEPENDENCY_LOST** message might be received at any time by a Processor Module that has declared itself dependent upon another Processor Module, if that Processor Module has, for any reason, experienced a problem. Data provided with the message serves to identify the Processor Module and the nature of the problem.

- **FILTER** messages are seen by Processor Modules that have declared themselves as ***Filters*** of another Processor Module's data records. Each data record produced by the other Processor Module is passed for examination to the Filter before it is stored in the current snapshot structure by the Engine. The Filter can inspect the record and either allow it to be stored or reject it.

Processor Modules are usually packaged in shareable images. (There is a provision for a Client Application to provide one Processor Module from the client's executable image.) A shareable image can contain multiple Processor Modules. (All Processor Modules shipped with TDC are in a single shareable image.)

The API defines an interface to be used by all Processor Modules. Example 1 illustrates some of the features of that interface.

```
TDC_Status_t XXX_PROCESSOR( TDC_CTX_t *APICtx,       // global context
                            TDC_ProcessorCTX_t *myCtx,//private context
                            TDC_Set_t *mySet, . . .) // record set
```

Example 1 – Interface to a Processor Module

Example 1 shows that a Processor Module receives three arguments (with a provision for optional arguments that is not discussed here) and returns a value of type TDC_Status_t. The first argument points to the global context for the operation; the second points to the Processor Module's private context; and the third usually points to a record set to be populated with data records by the Processor Module. The name for the function through which the Processor Module interacts with the Engine consists of a variable part, "XXX" in the example, followed by "_PROCESSOR." "XXX" is the name by which the Processor Module is known within the TDC system, and the name that a client application uses to specify that the Processor Module be loaded; "XXX_PROCESSOR" is the name by which the TDC Engine attempts to load it from a shareable image.

Example 2 shows how a Processor Module might respond to LOAD messages. In this case, the Processor Module inspects the global context to determine the type of operation to be performed. For a collection operation, the Processor Module indicates, through its private context the following: that it requires CMKRNL privilege, that it produces data records, that it should be called at all SNAPSHOT synchronization points, and that its correct operation depends on inclusion of another Processor Module (in this case, the "PRO" Processor Module supplied with TDC to produce Process metrics). If the Processor Module has dependencies on several Processor Modules, their names are supplied in a comma-separated list (for example, "PRO,DSK" if there were also a dependency on TDC's disk-utilization and performance-capturing Processor Module).

```
    case TDC_K_LOAD:
        if ( APICtx->TDC_CTX_Operation == TDC_K_Collect )
        {
            myCtx->TDC_PCTX_Privileges.prv$v_cmkrnl = 1;
            myCtx->TDC_PCTX_Capabilities = TDC_CAP_M_PRODUCER;
            myCtx->TDC_PCTX_Synchronization = TDC_SYNC_M_SNAPSHOT;
            myCtx->TDC_PCTX_DependsOn = "PRO";
        }
        return TDC_K_STATUS_Success;
```

Example 2 – Handling a LOAD Message

Example 3 illustrates the standard mechanism by which a Producer generates data records and provides them to the TDC Engine for storage. (Error-checking is omitted for brevity.)

```
#include “XXX.h”

case TDC_K_SNAPSHOT:
{
  XXX_Rec *data;
  data = (XXX_Rec *) APICtx->TDC_CTX_DataBuffer;
  for ( int k = 0; k < number-of-records; k++ )
  {
    memset( data, ‘\0’, sizeof *data );
    data->XXX_Prefix.TDC_REC_WU_RecSize = sizeof *data;
    data->XXX_Prefix.TDC_REC_WU_TypeC = XXX_Version;
    data->XXX_Prefix.TDC_REC_A_TypeA =
myCtx->TDC_PCTX_A_ProcessorDesc->TDC_PDESC_A_RecordID.TDC_RecType;
    collect_my_data( data, k );
    APICtx->TDC_CTX_RecordData( APICtx, myCtx, mySet );
  }
}
return TDC_K_STATUS_Success;
```

Example 3 – Storing Data

Each Producer should declare its data record and any other useful information in a *header file* so that the declarations are accessible to Consumers. In this case, the header is “XXX.h”, and the data record is declared in that header as type XXX_Rec. At the SNAPSHOT message, a pointer to the XXX_Rec type is declared and made to point to the record buffer supplied by the global context. All data records have a common prefix and, after clearing the data buffer, the record prefix is initialized with a record size, a record-version identification, and a record ID assigned by the TDC Engine that is copied from the processor module description for this Processor Module. A data record is then created and handed off to the Engine for storage through the TDC_CTX_RecordData function pointer in the global context. At that point, any Filters for this record type are invoked by the Engine to accept or reject the record before the Engine actually stores it in the current snapshot.

Example 4 shows how a Consumer might access the data provided by the “XXX” Processor Module shown in Example 3. Here, the Consumer uses the API to locate the record set header for “XXX” data records in the current snapshot. The API call is made through the TDC_CTX_FindInSnapshot pointer in the global context. The Consumer then uses the API to access each record in the record set by means of a series of calls through pointer TDC_CTX_AccessSetRecord in the global context.

Note that the “XXX.h” header file is included when compiling the Consumer shown in Example 4. The declaration of the record type from the header is used to declare an appropriate pointer for use in accessing the data records.

```
case TDC_K_CONSUME_DATA:
{
  XXX_Rec   *data;
  TDC_Set_t *XXX_Set;
  XXX_Set = APICtx->TDC_CTX_FindInSnapshot( APICtx,
                                            APICtx->TDC_CTX_CurrentSnapshot,
                                            “XXX” );
  for ( int k = 0; k < XXX_Set->TDC_SET_LU_RecordCount; k++ )
  {
    data = (XXX_Rec *) APICtx->TDC_CTX_AccessSetRecord( APICtx, XXX_set, k );
    if ( NULL != data )
        processData( data );
  }
}
return TDC_K_STATUS_Success;
```

Example 4 – Accessing Data

Client Applications

Client Applications need not be particularly complex. Example 5 shows a complete Client Application, with error-checking and most infrastructure omitted for brevity. It collects data over a 24-hour period, using separate data files for each hour's data.

```
TDC_CTX_t *APICtx;
char       filename[60];

APICtx = malloc( sizeof *APICtx );
TDC_INIT( APICtx );

for ( int k = 1; k <= 24; k++ )
{
  TDC_REGISTER( APICtx, "DSK", NULL, NULL );
  TDC_REGISTER( APICtx, "XXX", "dev:[dir]myshr", NULL );

  APICtx->TDC_CTX_Operation = TDC_K_Collect;
  APICtx->TDC_CTX_IntervalCount = 30;
  sprintf( filename, "%%N$DAILY-%%D-%.2d", k );
  APICtx->TDC_CTX_DataFile = filename;

  TDC_PREPARE( APICtx );
  TDC_START( APICtx );
  TDC_END( APICtx );
}

TDC_FINISH( APICtx );
free( APICtx );
```

Example 5 – A Client Application

In the example, memory is allocated for the global context, which is then initialized by calling the Engine's initialization function, TDC_INIT().

The Engine is then instructed, by calls to function TDC_REGISTER(), to load TDC-supplied Processor Module "DSK" and externally-provided Processor Module "XXX" from the shareable image at dev:[dir]myshr.exe. Recall from previous examples that "XXX" has a dependency on the "PRO" Processor Module; therefore, the "PRO" module is loaded automatically from the TDC library by the Engine.

Several parameters to define and control the current operation are then specified: it is to be a collection operation that runs for 30 2-minute (the default) intervals (one hour), producing 31 system snapshots. Data collected during the first hour will be stored in file NODE$DAILY-041119-01.TDC$DAT in the current default directory; the second hour's data will be placed in file NODE$DAILY-041119-02.TDC$DAT, and so on. Note that the Engine recognizes three special "escape characters" in file specifications: %N is replaced by the system's node name ("NODE" above); %D is replaced by the current date, or by the start-date specified for the operation, in YYMMDD format (for example, a date of 19-NOV-2004 is placed into the file specification as "041119"); and %Tis replaced by the current time, or by the start time specified for the operation, in HHMMDD format (for example, 15:30:10 is placed into the file specification as "153010").

The Engine is next initialized for the current operation by the call to TDC_PREPARE(), after which the Engine is given control until the operation completes by the call to TDC_START(). A Client Application can retain control over the operation by making a series of calls to TDC_COLLECT_SNAPSHOT() to collect data one snapshot at a time, rather than handing control over to TDC_START().

When the operation completes, TDC_END() is called to free all resources used during the collection. The final status of the operation and other information is available to the Client Application in the global context after TDC_START() returns and before TDC_END() is called.

Because TDC_END() destroys all context information, the Engine must be fully initialized for each operation within the loop; only a single initialization of the global context is required.

When all collections have completed, TDC_FINISH() is called to free any resources still held by the TDC Engine.

Software Developers Kit

The downloadable kit available at the URL specified at the end of this article contains the Software Developers Kit (SDK). The SDK includes header files for all TDC data records (the only documentation for the content and format of those records); a header file with the API's common declarations; complete source code for the TDC control application; sample Client Application code; sample Processor Module code; a "stub" to use as a starting point in developing Processor Modules; miscellaneous support files; and a TDC Programmer's Guide. The Programmer's Guide can also be downloaded separately from the web site.

In planning any development work, consider the information provided by "System Management Overview" below and, in particular, heed the cautions under "Building Software that Uses TDC."

The TDC API

The API is fully documented in the *TDC Programmers Guide* and by comments in the files included with the SDK. Of particular note among the files is TDC_COMMON.H, which declares all the data structures used when interacting with the API. Table 1 below briefly describes the functions that comprise the API.

The names listed under "From Processor Modules" are the names of function pointers in the global context through which Processor Modules make API calls (Processor Modules should never call API functions by name). If there is no entry for a function under "From Processor Modules," then that function is not suitable for use by a Processor Module.

Names listed under "From Client Applications" are the entry names though which Client Applications can make API calls; a Client Application is also free to make API calls through the function pointers listed under "From Processor Modules."

From Client Application	From Processor Modules	Description
Control Functions		
TDC_INIT		Initializes the API.
TDC_REGISTER		Informs the Engine about a Processor Module that should be loaded for the current operation.
TDC_PREPARE		Initializes the Engine for the current operation.
TDC_START		Performs the current operation.
TDC_COLLECT_SNAPSHOT		Performs one step of a collection operation.
TDC_READ_SNAPSHOT		Performs one step of an extraction operation.
TDC_CONTINUE		Transitions from step-by-step processing mode to surrendering control to the TDC Engine to continue and complete the current operation.
TDC_END		Completes the current operation.
TDC_FINISH		Completes use of the API.
TDC_HALT_OPERATION	TDC_CTX_HaltOperation	Requests that the current operation halt (an unscheduled halt).

From Client Application	From Processor Modules	Description
TDC_TEST_COLLECTION_STATUS	TDC_CTX_TestCollectionStatus	Tests whether the current collection operation has been requested to halt.
Snapshot Manipulation Functions		
TDC_NEW_SNAPSHOT		Creates a new snapshot structure
TDC_RESERVE_SNAPSHOT	TDC_CTX_ReserveSnapshot	Prevents freeing of a snapshot structure, to permit continued access to it.
TDC_RELEASE_SNAPSHOT	TDC_CTX_ReleaseSnapshot	Allows freeing of a snapshot structure previously "reserved."
TDC_FIND_IN_SNAPSHOT	TDC_CTX_FindInSnapshot	Locates the record set for a data type within a snapshot structure.
Record Set Manipulation Functions		
TDC_ACCESS_RECORD_SET	TDC_CTX_AccessRecordSet	Locates the "k^{th}" record set within a snapshot structure.
TDC_ACCESS_SET_RECORD	TDC_CTX_AccessSetRecord	Locates the "k^{th}" record within a record set.
TDC_NEW_RECORD_SET	TDC_CTX_NewRecordSet	Creates a new record set.
TDC_CLONE_RECORD_SET	TDC_CTX_CloneRecordSet	Creates a "clone" of a record set.
TDC_NEW_RECORD_SET_IN_SNAPSHOT	TDC_CTX_NewRecordSetInSnapshot	Replaces one record set in a snapshot with another.
TDC_FREE_RECORD_SET	TDC_CTX_FreeRecordSet	Frees memory used by a record set.
Processor Module Manipulation Functions		
TDC_GET_RECORD_TYPE	TDC_CTX_GetRecordType	Gets the ID number assigned to a record type for the current operation.
TDC_GET_RECORD_NAME	TDC_CTX_GetRecordName	Gets the name of the record type associated with an ID number.
TDC_CALL_PROC_MODULE	TDC_CTX_CallProcModule	Allows indirect calls to a specific processor module.
TDC_GET_INFO	TDC_CTX_GetInfo	Returns information about the Engine state (38 items) or about Processor Modules (37 items).
TDC_SET_ATTRIBUTES	TDC_CTX_SetAttributes	Allows modification of some operational parameters of a Processor Module (11 items)
Data Record Manipulation Function		
	TDC_CTX_RecordData	Gives a data record to the Engine for storage.
File-Oriented Functions		
	TDC_CTX_OpenCollection	Opens a collection file.
	TDC_CTX_CloseCollection	Closes a collection file.
	TDC_CTX_WriteCollectionRecord	Writes a data record into a collection file.
	TDC_CTX_ReadCollectionRecord	Reads a data record from a collection

From Client Application	From Processor Modules	Description
		file.
TDC_CREATE_FILE_NAME	TDC_CTX_CreateFileName	Creates a file spec, including substitutions for %N, %D, and %T.
Utility Functions		
TDC_GET_TIME_DIFF_SECS	TDC_CTX_GetTimeDiffSecs	Calculates the difference in seconds between two timestamp values.
	TDC_CTX_DisplayTime	Returns a text string that represents a timestamp value.
	TDC_CTX_TraceLog	Produces tracing and logging messages.
	TDC_CTX_WriteFormatted	Writes formatted output to a file.

Table 1 – TDC API Functions

System Managment Overview

TDC V1.0

You might have installed TDC Version 1.0 for use with a third-party system management solution. TDC Version 2.1 is not compatible with software that makes use of TDC Version 1.0. However, installation of TDC Version 2.1 does not remove files installed by TDC Version 1.0, nor does it impact your ability to continue using the TDC Version 1.0 software with your third-party system management solution. If you rely on software that makes use of TDC Version 1.0, do not remove TDC Version 1.0 from your system until your third-party software has been updated to support use of TDC Version 2.1.

Version 2.1 Files

The Performance Data Collector consists of a control application, an execlet, two shareable images, and various support files. To enable installation in various types of shared-system environments, image file names reflect both the platform for which each is intended and a TDC build identification. Thus, TDC$APISHR$I_V820-0070.EXE is an image for Integrity Server (I64) systems ("$I") running OpenVMS Version 8.2 ("_V820") and provided by the installation of TDC Version 2.1-70 ("-0070").

Version 2.1 Kit Variants

For logistical reasons, two distribution variants of the TDC software exist: **TDC_RT** for the software installed with OpenVMS, and **TDC** for the software downloaded from the web site (URL at the end of this article). TDC_RT contains the software required to run TDC on a specific platform, while TDC contains software to run TDC on any supported platform. Assuming a common TDC build identification (*for example,* Version 2.1-70), there is no functional difference between the software installed by TDC_RT and the software that is installed on the same system by TDC.

The downloadable TDC kit will be revised as needed to deliver maintenance updates and functionality enhancements, while the TDC_RT variant will be updated only with new OpenVMS releases. Installation of an updated TDC variant might or might not remove files previously installed by TDC_RT, depending on your system configuration. Since both variants utilize the same startup file (TDC$STARTUP.COM), which initializes the system environment to use the most recent TDC images available on the system, an older TDC_RT installation and a newer TDC installation can safely co-exist. Do not remove TDC_RT from your system, even if you have installed an updated TDC variant.

Installing TDC in an OpenVMS Cluster

By default, both TDC_RT and TDC install files into SYS$COMMON:[TDC]. Your cluster environment might be set up such that SYS$SYSROOT applies first to a node-specific root (for example,

SYS$SPECIFIC:), then to a shared root (for example, SYS$COMMON), and finally to a clusterwide root (for example, CLU$COMMON). The downloadable TDC kit can be safely installed into the most widely-shared root (for example, CLU$COMMON) that is convenient, provided that installations of the downloaded TDC kit are always performed from the same cluster node so that the OpenVMS product database is consistent.

Building Software that Uses TDC

For the reasons cited above, an individual system might have multiple versions of the TDC software installed. That should not present a problem if you follow a few guidelines in developing and deploying your software:

- Make sure that the SYS$STARTUP:TDC$STARTUP.COM has been run before you build or run your application.
- Always access the TDC software through one of the logical names defined by TDC$STARTUP.COM (TDC$APISHR for the Engine; TDC$LIBSHR for the library of Processor Modules), rather than by specific file names. This will prevent possible problems with multiple simultaneous activations of different TDC images within your application.
- For the same reason, do NOT link shareable images that contain your own Processor Modules against the TDC Engine (TDC$APISHR).
- Link Client Applications against the TDC Engine (TDC$APISHR), but, in general, not against shareable images containing Processor Modules.

Running the TDC Control Application

You can use the Performance Data Collector control application, TDC$CP, to process data files and for various administrative tasks, as well as to collect data. Unless you are running the TDC control application, TDC is not actually "running" on a system. (In other words, it does not run in the background just because OpenVMS is running).

Data collection and some administrative tasks require you to enable various privileges before running the application. The full set of privileges required to perform all tasks and collect all data includes CMKRNL, LOG_IO, NETMBX, PHY_IO, SYSLCK, SYSPRV, and WORLD; SYSNAM privilege is required to run the startup procedure, SYS$STARTUP:TDC$STARTUP.COM. The control application is not installed with the required privileges; HP strongly recommends that you do not install it as a privileged image.

Before starting the control application, each user must define the "TDC" command as shown in Example 6.

```
$ @SYS$STARTUP:TDC$STARTUP ! if not done at system boot
$ SET COMMAND SYS$COMMON:[TDC]TDC$DCL
```

Example 6 – Defining the TDC command

The user can then either run the control application interactively, by typing just TDC at the DCL prompt, or extend the TDC DCL command with a TDC command and qualifiers to specify a complete operation. In the latter case, control is returned to DCL after the operation completes. In interactive mode, you can enter a series of TDC commands before exiting the control application.

Once running, the control application can run collection operations in detached processes, on the local node, on specified nodes within the cluster, or on all nodes within the cluster. The control application can also be used to identify running collections, obtain status from running collections, or stop running collections anywhere within the OpenVMS Cluster.

For more information

For general TDC support issues, use standard OpenVMS support channels.

For software-development questions and issues (only), send email to VMSTDCV2@hp.com.

Software updates and documentation are available for free download at:
http://h71000.www7.hp.com/openvms/products/tdc/

Note: The software kit available at the web site contains runtime software for all supported OpenVMS platforms as well as the Software Developers Kit (SDK). The software kit is packaged one way for downloading and unpacking on Alpha platforms, and a different way for downloading and unpacking on Integrity Server platforms. The contents of the two packages are identical; in particular, both packages provide TDC software for both Alpha and Integrity Server environments. The PCSI-based installation procedure allows you to select the runtime environments you want to install and whether or not to install the SDK.