The Impact of Test Instrumentation with Distributed Processing Capabilities on Test Program Set (TPS) Architecture and Development

Michael McGoldrick
Assembly Test Division
Teradyne, Inc.
North Reading, MA USA
michael.mcgoldrick@teradyne.com

***Abstract*—Modern digital interconnection methods have placed significant computational demands on computers controlling test systems, and adding dedicated computing resources for high performance digital test instrumentation to the test system can help meet these demands. This paper examines the impact of these additional computing resources on the design of a TPS, and proposes a software framework to assist developers in creating TPSs for multi-computer environments.**

Keywords-TPS; digital; LVDS; synchronization; message passing; instrument driver; inter-process communication; subTPS; IVI; IVI custom specific driver; clock speeds; LXI

I. INTRODUCTION

As the performance of digital interconnection technology has grown over time, the demands on test instrumentation have followed suit. Parallel digital connections that used to use standard logic families (e.g. TTL, CMOS, ECL, etc.) and operated at speeds of up to a few tens of MHz, have made way for Low Voltage Differential Signaling (LVDS) devices operating at hundreds of MHz. Similarly, serial digital connections have moved from the simple RS-232 application operating at perhaps hundreds of kHz to embedded clock applications like PCI Express [1, pp 453-486] or Fibre Channel [2] operating at speeds in the several-GHz region over copper or optical fiber. The increased speed of these interconnects has impacted test instrumentation design in several ways:

1. The higher speed operation, particularly for parallel digital connections, can place restrictions on the maximum distance from the test instrumentation and a unit under test (UUT).

2. Higher speed operation can require high data throughput in order for test instrumentation to keep up with UUT operational demands.

3. The higher speed operation of the UUT is often the result of a need to pass a very large amount of data between subsystem components. This can require that the test instrumentation be able to store and process large quantities of data.

A prime example of the effect on higher speed operation on test instrumentation to UUT distance is the PCI bus, which uses reflected wave switching and requires bus end points to be close enough to guarantee no more than 10 ns of out and back propagation time from device to device when operating at clock speeds of 33 MHz. At 66 MHz clock speeds, this value drops to 5 ns [3].

UUT to test instrumentation distance is also an important consideration for test applications where the test instrumentation is emulating the expected end environment for the UUT, and needs to react in real time to events initiated by the UUT and measure the UUT's response to test instrumentation generated events. If the distance between the UUT and test instrumentation is larger than the distance between the UUT and the system components in the end environment, the extra propagation time can result in the UUT being tested at an artificially slow operational speed and prevent verification of the UUT's operation at its intended rate. If the UUT requires timely test instrumentation responses to UUT events to even function properly, large distances between the UUT and the test instrumentation may preclude the UUT from being tested at all.

The switch from parallel to serial buses, as in the case from PCI to PCIe, has increased the rate of information flow over those buses, as indicated in Fig.1.

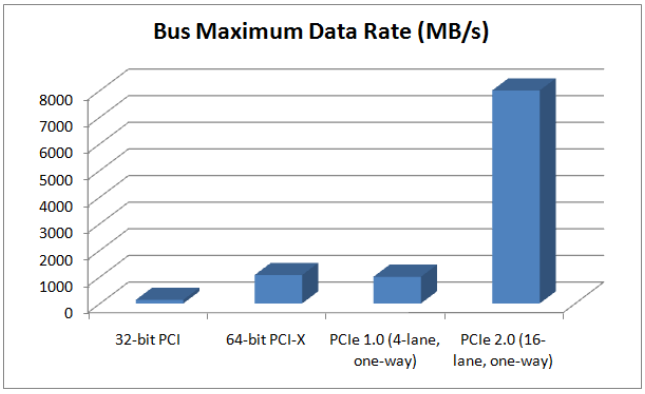

Figure 1. Data rate comparisons of different PCI bus implementations

978-1-4244-9363-0/11/$26.00 ©2011 IEEE

The rates for PCIe 3.0 are anticipated to be even higher. While test instrumentation can use the same physical layer devices as the UUT for driving and receiving on the bus, the challenge to test instrumentation designers is to provide the infrastructure to process or generate data at those rates.

The data rate increases are made even more challenging by the sheer bulk of data to be sent or received by the test instrumentation. One second of source data for a video digital to analog converter like the Texas Instrument THS8135 [5] requires approximately 960 MB of data, while capturing a second of data from a high speed analog to digital converter like the National Semiconductor ADC10DV200 [6] requires storage capacity of perhaps 800 MB.

One way of addressing these challenges is to create high performance digital (HPD) test instrumentation located physically closer to the UUT, and to back up that test instrumentation with sufficient computing and data bandwidth resources to meet the UUT test requirements. The remainder of this paper explores the impact of this test instrument architecture on TPS design and execution.

II. Dual-Computer Architecture

Computers that control test systems today need to manage multiple tasks, including hosting the test executive software that runs the TPS and displays an operator's or developer's graphical user interface to control test execution and TPS/UUT debug. In addition to HPD instrumentation, test systems also contain other test assets like power supplies, meters, oscilloscopes, switches, RF instrumentation, analog stimulus and measurement instrumentation, etc., all of which need to be controlled during TPS execution. Test results data may also need to be archived over a company's LAN. Diagnostics resources (schematics, flow-charts, etc.) may need to be consulted. These tasks can limit the computing and data bandwidth resources available to HPD test instrumentation. A way to help ensure that the HPD instrumentation has sufficient computing resources to meet its high performance test objectives is to control it with a separate computer, and to off-load the HPD-related tasks to that computer.

While adding an embedded processor to an instrument is certainly an option, this approach may not be suitable in all test applications. For example, digital test instrumentation required in some parallel test applications may not fit on a single instrument, requiring some sort of cooperative execution between multiple embedded processors most likely managed by the test system's computer. Or the HPD instrumentation may need to operate in tight cooperation with other test instrumentation, again requiring some sort of test system/embedded processor interaction. In both of these situations, the test system computer is still taking an active, although diminished, role in managing the HPD resources. A perhaps more significant issue with the embedded processor approach is the increased complexity of the TPS development effort. Embedded application development uses a different toolset than typical TPS development does, requiring TPS developers to have specialized software skills and complicating the TPS debugging effort.

A different way to approach the problem of HPD computing resources is to control the HPD instrumentation with a separate PC that is networked with the computer running the rest of the test system instrumentation. In the remainder of this paper, the original computer is referred to as the "test station computer", and the computer controlling the HPD instrumentation is referred to as the "instrumentation computer". In a test system that has such a dual-computer architecture as shown in Fig. 2, the instrumentation computer runs the code that controls the HPD instrumentation, and the test station computer controls the rest of the test assets and performs the other test-related tasks discussed above.

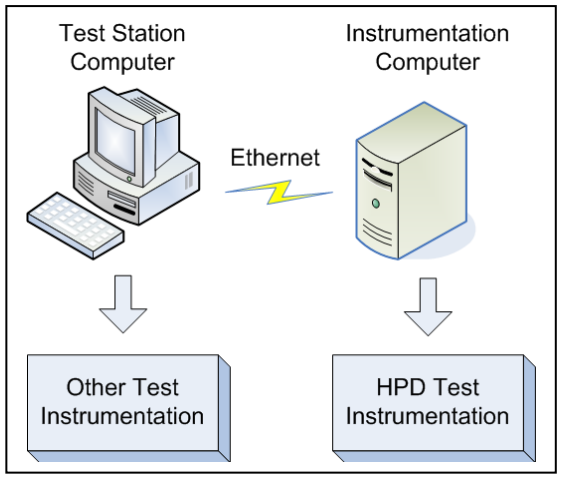

Figure 2. Dual computer test system architecture

In effect, the HPD instrumentation control code that is normally executed with the other test instrumentation control code in the TPS is extracted and moved from the test station computer to the instrumentation computer. The extracted code is executed as part of some other secondary program, which can be developed using the same toolset that is used to develop the TPS. In its place, the TPS contains code to initiate execution of the secondary program with the HPD instrumentation control code on the instrumentation computer.

A. Application Driver

In its simplest embodiment, the revised TPS code initiates some action on the instrumentation computer, and then waits for that action to complete before continuing its own execution, much as a simple function call in a C program behaves. To do this, the TPS code needs to initiate an action on a different computer using some sort of inter-process communication mechanism. One approach would be to make use of system calls exported by the operating system on the test station computer. That has the drawbacks of making the TPS operating-system dependent and difficult to transport, and forces the TPS developer to become expert in the use of these system calls.

A better approach would be for the HPD instrument vendor, in addition to the expected instrument driver, to provide a second driver that is responsible for controlling the application running on the instrumentation computer. This so-called

application driver hides the operating system specific details of initiating actions on the instrumentation computer.

B. The subTPS

The TPS developer still needs to include the HPD instrumentation code that is executed on the instrumentation computer in some executable software component that is run on the instrumentation computer. This component makes calls in the HPD instrument driver to establish a session with an HPD instrument, issues setup instructions, initiates measurements or tests, processes the results, and terminates the session. Due to the similarities between the instrumentation computer component code and the main test program code, it is natural to think of the instrumentation computer component as being a TPS in its own right, one that works cooperatively with the main test program running on the test station computer. For the remainder of this paper, the term "TPS" refers to the main test program running on the test station computer, and the term "subTPS" refers to the component running on the instrumentation computer, and serves to highlight that a subTPS runs under the direction and control of a TPS.

Since the test executive running the TPS is generally responsible for interfacing with the operator, the subTPS rarely needs to graphically display output to the operator or process mouse or keyboard events. Therefore, a subTPS needs to be nothing more than a simple console application (although other application types like graphical applications or test executives running on the instrumentation computer are also possible). Console applications have the advantage of being easy to develop, include command line parameters to affect their execution, and return exit codes or status information when the application terminates.

C. Client/Server Architecture

While programmatically starting programs locally (i.e. on the same computer) is simple and straightforward, initiating processes on a remote computer is more difficult due to security concerns. One way to circumvent this is to use a client/server approach, where a trusted client on the test station computer relays a request to a server component on the instrumentation computer to run a subTPS. Using this technique, the subTPS process creation request is done locally by the server component, which monitors the execution of the initiated subTPS and returns its exit code to the TPS when it terminates. The TPS developer makes a request for the instrumentation computer to execute a subTPS by calling a function exposed by the application driver.

III. TPS/SUBTPS INTERACTION

In the simple TPS/subTPS interaction described in Fig. 3, subTPS operation completion is signaled when the subTPS process exits. The process exit code can be used to indicate status information to the TPS.

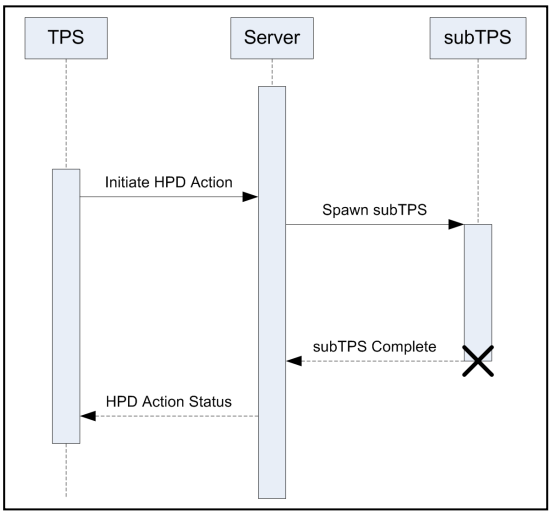

Figure 3. Sequence diagram depicting interaction between a TPS and a subTPS

This particular operational model creates a subTPS process on each application driver call. If the TPS initiates several of these operations, this can have the following disadvantages:

- The operating system overhead for creating a new process is incurred on every application driver call
- There is no convenient way to maintain state information between multiple application driver function calls

Another approach that does not have these disadvantages is to create a subTPS that performs a series of tasks one at a time at the request of the TPS. To support this, some other mechanism besides process termination must be used to indicate completion of a particular task.

A. Synchronization Objects

Modern operating systems provide multiple ways of synchronizing activity between processes either on the same computer or running remotely, through such resources as events, semaphores, and mutexes. As when creating subTPS processes, it is desirable to hide calls to these process synchronization operating system functions within application driver functions. A convenient abstraction for this functionality is a "synchronization object", which exposes methods to set the object state to "reset" or "signaled", and provides a method that blocks further program execution until the synchronization object is in the "signaled" state.

Synchronization objects provide the necessary functionality for implementing a subTPS that performs a series of tasks one at a time in response to requests from a TPS. The sequence diagram in Fig. 4 illustrates how a TPS and a subTPS interact with a synchronization object to achieve the desired operational behavior.

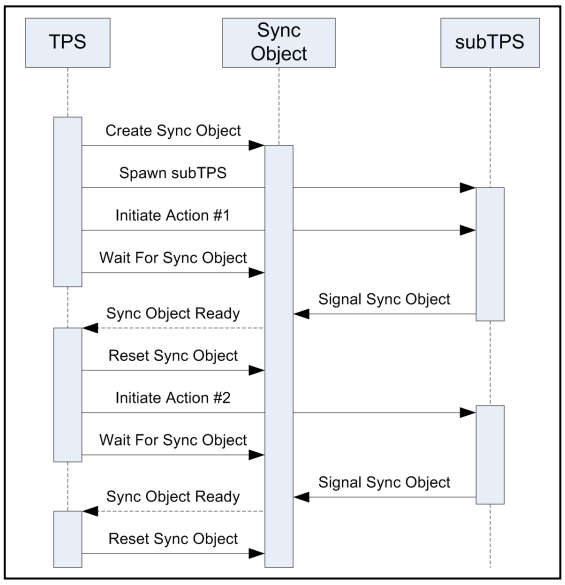

Figure 4. Synchronization object acting to control TPS/subTPS execution

Alternatively, the TPS can initiate some action by the subTPS and then proceed to run in parallel with the subTPS. In this case, a synchronization object can be used to ensure that some required subTPS activity is complete before TPS execution continues, as shown in Fig. 5.

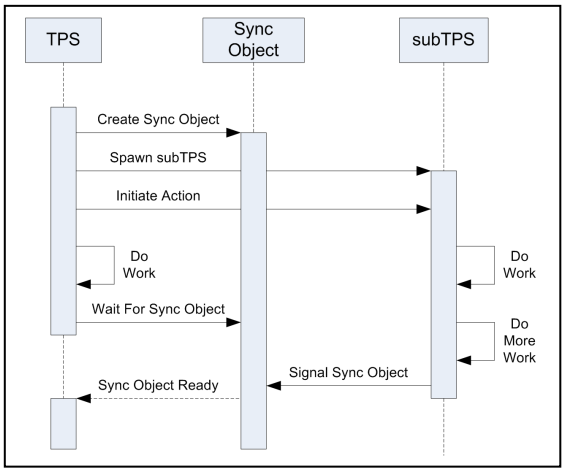

Figure 5. Synchronization object used to join parallel execution of the TPS and subTPS

Earlier, we described using the process exit code of a subTPS to report status information to the TPS. To maintain this functionality using synchronization objects, the signaling of a synchronization object is also accompanied by an integer data value that serves the purpose that the subTPS process exit code previously did. When the signal is received by the TPS, the TPS can test this value to determine the result of the initiated operation.

B. subTPS Services Driver

One issue that has not been addressed is how a subTPS references a synchronization object that is created by the TPS, which is running on a different computer. One approach is to have the server component actually manage the synchronization object. The TPS receives signals of these objects via the existing network connection between the application driver and the server component. The subTPS, on the other hand, is a separate process from the server component. To be able to reference the synchronization object managed by the server component, the subTPS must also become a client of the server component. Just as the TPS gains access to the server component via an application driver, access to the server component from the subTPS is provided by a subTPS services driver. This driver provides the function that the subTPS calls to signal the synchronization object, for which the TPS is waiting.

Fig. 6 depicts how the TPS, subTPS, and the various supporting software components relate to each other, at least in block diagram form.

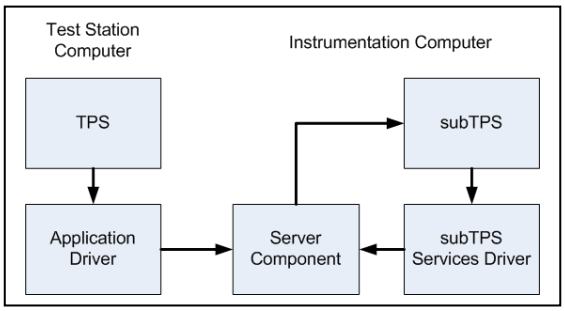

Figure 6. TPS, subTPS, and supporting software components

IV. MESSAGING

In the previous discussion about synchronous operation, the series of actions undertaken by the subTPS were in the form of a simple sequence. An alternative approach that provides more flexibility is message passing. In this scheme, the subTPS uses the subTPS services driver to register a handler function that is executed in response to an incoming message from the TPS, which sends the message using a function in the application driver. The message consists of a context value, which allows the handler to quickly determine the meaning of the message, and an arbitrarily sized payload of binary data. One usage model would be for the message context value to specify a command and the payload the command arguments.

A subTPS structure that implements the messaging approach might have the structure indicated in Fig. 7. The subTPS starts up, registers a message handler, and then waits for a signal. When incoming messages are received from the TPS, the handler is called and the message is processed. A message with an appropriate context value (in the code example, this is represented by the literal ‘QUIT’) signals the synchronization object, and allows the main function to continue and the subTPS process to exit.

Messages can not only flow from the TPS to the subTPS but also the other way around: the TPS can register a handler function that is executed in response to an incoming message from a subTPS, which sends the message using a function in the subTPS services driver.

```
int main(int argc, char ** argv)
{
    …
    RegisterMessageHander(MyHandler);
    …
    // Blocks until "Foo" is signaled.
    // Prevents subTPS from exiting prematurely.
    WaitForSyncObject("Foo");

    return status;
}

void MyHandler(
    int context, int byteSize, byte *payload)
{
    switch (context) {
    case CMD1:
        <parse payload for CMD1>
        break;
    case CMD2:
        <parse payload for CMD2>
        break;
    …
    case CMDN:
        <parse payload for CMDN>
        break;
    case QUIT:
        SignalSyncObject("Foo");
        break;
    default:
        <handle unsupported command>
        break;
    }
}
```

Figure 7. Generic message handler structure in the subTPS

Messages provide a more powerful means of transferring data between a TPS and a subTPS than using command line parameters and process exit codes. However, care must be taken to ensure that the TPS and subTPS interpret the data in the same way. If the TPS and subTPS are both written in C or C++, you can create a common header file, in which payload structures are defined, and form and extract payloads as instances of those payload types. For example, consider a simple example of passing an instance of a complex number implemented in a C struct as shown in Fig. 8.

```
struct complex {
    double magnitude;
    double phase;
};
```

Figure 8. Complex number data structure definition

This data type and possibly other items that are common to the TPS and the subTPS are then defined in a header file, *CommonTPSTypes.h*, whose contents are shown in Fig. 9.

```
// CommonTPSTypes.h
// Contains common types for TPS and subTPS

// Definition of complex type
struct complex {
    double magnitude;
    double phase;
};

// Command list
#define QUIT               0
#define SEND_COMPLEX_VALUE 1

etc.
```

Figure 9. CommonTPSTypes.h header file contents

The TPS references the *CommonTPSTypes.h* header in its code. Sample code to send a complex number value to a subTPS is shown in Fig. 10. This shows a hypothetical application driver function *SendMessageToSubTPS* being called with a context value of SEND_COMPLEX_VALUE (defined in *CommonTPSTypes.h*) and the complex number value as the message payload.

```
#include "CommonTPSTypes.h"

void SendData(struct complex complexValue)
{
    SendMessageToSubTPS(
        subTPSHandle,
        SEND_COMPLEX_VALUE,
        sizeof(struct complex),
        &complexValue);
}
```

Figure 10. Sample code to send a message to a subTPS

Fig. 11 shows representative subTPS code that receives the message and extracts the complex number value from the payload. The subTPS references the same *CommonTPSTypes.h* header file, which gives it access to the SEND_COMPLEX_VALUE context value and the complex number structure.

```
#include "CommonTPSTypes.h"

int main(int argc, char ** argv)
{
    …
    RegisterMessageHander(MyHandler);
    …
    // Blocks until "Foo" is signaled.
    // Prevents subTPS from exiting prematurely.
    WaitForSyncObject("Foo");

    return status;
}

void MyHandler(
    int context, int byteSize, byte *payload)
{
    switch (context) {
    case SEND_COMPLEX_VALUE:
        if (byteSize == sizeof(struct complex))
        {
            struct complex complexValue =
                *(struct complex *) payload;
            // Do something with value
        }
        break;
    case QUIT:
        SignalSyncObject("Foo");
        break;
    }
}
```

Figure 11. Sample code to receive a message from the TPS

By defining a payload structure in a header file and referencing that header file in both the TPS and subTPS, it is possible to send messages with payloads of any organization back and forth between a TPS and a subTPS.

V. SUGGESTED IMPLEMENTATION

The application driver, server component, and subTPS services driver are software components that operate cooperatively to support remote control of HPD instrumentation from a TPS. While these components do not communicate with instrumentation, the application driver and subTPS services driver should nonetheless have the same look and feel as the instrument drivers in the TPS to make those programming interfaces easy for TPS developers to utilize. As most modern instrument drivers are IVI-based, an IVI look and feel seems natural to use for these components. At a minimum, the programming interfaces should support IVI data types and follow IVI function naming conventions. The server component's programming interface has no such constraint, since neither the TPS nor the subTPS communicate with it directly.

A. Application Driver

In one embodiment of the dual-computer architecture, the instrumentation computer and the instrumentation that it controls can constitute an LXI instrument, with which the test station computer communicates over a local area network. Such an "instrument" (in reality, here, an entire subsystem) is required to have an IVI driver [7]. In this case, the application driver not only has an IVI look and feel – it is an actual IVI driver, most likely an IVI custom specific driver.

IVI custom specific drivers are required to expose certain functions in order to comply with the standard [8]. The recommended functionality required of a C language application driver is above and beyond this IVI-prescribed interface and includes functions listed in table I. Each of the functions listed in the table are assumed to return ViStatus values, and all function names have a prefix, as prescribed by the IVI standard [9] for IVI-C programming interfaces.

TABLE I. APPLICATION DRIVER FUNCTIONS

Function Prototypes (return type and prefixes omitted)	Description
Load(ViSession vi, ViConstString applicationName, ViConstString remotePath, ViInt32 *applicationHandlePtr)	Locates an executable program on the instrumentation computer, and gives it a handle.
Start(ViInt32 applicationHandle, ViConstString cmdLine)	Starts the executable with the given handle, passing the specified command line parameters.
Wait(ViInt32 applicationHandle, ViReal64 timeout, ViInt32 *exitCodePtr)	Pauses execution until the executable with the given handle completes execution. The function fills in the subTPS exit code.
RegisterMessageCallback(ViSession vi, MessageFromApplicationHandler callback)	Registers a callback function, which is invoked when a subTPS sends a message.
SendMessage(ViSession vi, ViInt32 applicationHandle, ViInt32 messageContext, ViInt32 payloadSize, ViByte * payload)	Sends a message to the executable with the given handle. The message includes a context value and a payload.
CreateSyncObject(ViSession vi, ViConstString syncObjectName, ViInt32 *syncObjectHandlePtr)	Creates a synchronization object with the specified name, and provides a handle to it.
OpenSyncObject(ViSession vi, ViConstString syncObjectName, ViInt32 *syncObjectHandlePtr)	Opens a previously created synchronization object with the specified name, and provides a handle to it.
DeleteSyncObject(ViSession vi, ViConstString syncObjectName)	Deletes a previously created synchronization object.
SignalSyncObject(ViSession vi, ViInt32 syncObjectHandle, ViInt32 contextValue, ViBoolean autoReset)	Sets the state of the synchronization object with the given handle to the signaled state, and passes the given context value to the program that is waiting for the object to be signaled. After the signal is received, the object is can be automatically returned to the reset state or left signaled.
ResetSyncObject(ViSession, ViInt32 syncObjectHandle)	Sets the state of the synchronization object with the given handle to the reset state.
WaitForSyncObject(ViSession vi, ViInt32 syncObjectHandle, ViReal64 timeout ViBoolean autoReset, ViInt32 *contextValuePtr)	Pauses execution of the TPS until the synchronization object with the given handle signals or the specified timeout elapses. The context value associated with the signal is filled in. After the signal is received, the object is can be automatically returned to the reset state or left signaled.

B. subTPS Services Driver

The purpose of the subTPS services driver is to provide a subTPS with indirect access to the TPS that spawned it to

deliver messages and to access synchronization objects created by the TPS. Since a running subTPS instance is only associated with a single TPS, the driver does not require any sort of session handle parameter in any of its exposed functions. As a result, the driver is not IVI-compliant, but should still use IVI data types to maintain a common look and feel with the actual instrument drivers referenced in a subTPS. Table II lists the recommended set of subTPS services driver functions, again assuming a C language implementation. Like the application driver, the return type and function prefixes are not shown.

TABLE II. SUBTPS SERVICES DRIVER FUNCTIONS

Function Prototypes (return type and prefixes omitted)	**Description**
SendMessage(ViInt32 messageContext, ViInt32 payloadSize, ViByte * payload)	Sends a message to the TPS. The message includes a context value and a payload.
RegisterMessageCallback(MessageFromTestStationHandler callback)	Registers a callback function, which is invoked when a subTPS receives a message.
OpenSyncObject(ViConstString syncObjectName, ViInt32 *syncObjectHandlePtr)	Opens a previously created synchronization object with the specified name, and provides a handle to it.
SignalSyncObject(ViInt32 syncObjectHandle, ViInt32 contextValue, ViBoolean autoReset)	Sets the state of the synchronization object with the given handle to the signaled state, and passes the given context value to the program that is waiting for the object to be signaled. After the signal is received, the object is can be automatically returned to the reset state or left signaled.
ResetSyncObject(ViInt32 syncObjectHandle)	Sets the state of the synchronization object with the given handle to the reset state.
WaitForSyncObject(ViInt32 syncObjectHandle, ViReal64 timeout ViBoolean autoReset, ViInt32 *contextValuePtr)	Pauses execution of the subTPS until the synchronization object with the given handle signals or the specified timeout elapses. The context value associated with the signal is filled in. After the signal is received, the object is can be automatically returned to the reset state or left signaled.

VI. CONCLUSION

In comparison to a TPS developed for a single computer system, developing a dual-computer-ready TPS necessarily involves writing code to deal with communicating between processes on different computers. By wrapping this functionality in familiar looking drivers, and hiding the system-specific details concerning inter-process communication across a local area network in lower-level software components, TPS developers can more readily and efficiently create these TPSs using tools, with which they are already familiar. The extra effort involved to make the cross-process calls is incremental, and the TPS developer's main focus remains on programming the HPD instrumentation, not on how to control the execution of the two programs in parallel.

REFERENCES

[1] Mindshare, Inc., R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture*, Boston, MA: Addison-Wesley, 2003.

[2] R. Kembel, *Fibre Channel A Comprehensive Introduction*, 2nd ed., Tuscon, AZ: Northwest Learning Associates, Inc., 2003, pp 69-78.

[3] Mindshare, Inc., T. Shanley, D. Anderson, *PCI System Architecture, Fourth Edition*, Boston, MA: Addison-Wesley, 1999, pp 299-308.

[4] *PCI Express® Base Specification Revision 2.1*, PCI-SIG, March 4, 2009.

[5] Texas Instruments, Inc. "THS8135 Triple 10-Bit, 240 MSPS Video DAC with Tri-Level Sync and Video (ITU-R.BT601)-Compliant Full Scale Range", Internet: http://focus.ti.com/lit/ds/symlink/ths8135.pdf, June 2002 [June 20, 2011].

[6] National Semiconductor, "ADC10DV200 Dual 10-bit, 200 MSPS Low-Power A/D Converter with Parallel LVDS/CMOS Outputs from the PowerWise® Family", Internet: http://www.national.com/pf/DC/ADC10DV200.html#Overview, June 20, 2011 [June 20, 2011].

[7] *LXI Standard*, LXI Consortium, Inc., Rev. 1.3, October 30, 2008.

[8] *IVI-3.2: Inherent Capabilities Specification*, IVI Foundation, Revision 2.1, April 15, 2011.

[9] *IVI-3.4: API Style Guide*, IVI Foundation, Revision 2.0, June 9, 2010.