Enabling Communication between ATLAS Environment with an Integrated Subsystem

Alan Kin
Assembly Test Division
Teradyne, Inc.
North Reading, MA USA
alan.kin@teradyne.com

***Abstract*—Technology upgrades bring about the need for fielded ATLAS-based environments to expand tests to include modern high speed digital systems. The ATLAS test language does not contain innate capability to describe new tests, so TPS developers use NAM or FEP constructs as the mechanism to communicate with an external system. An external system, such as a self-contained, fully-integrated instruments within a PXIe Express chassis that includes a dedicated PC controller, is available to provide such high speed serial bus tests. While these subsystems can be integrated with an existing test system that has a Windows-based computer, additional work is needed to enable the integrated subsystem to communicate with the ATLAS operating environment.**

The ATLAS Host API was developed to address both needs; new or existing TPS's in the ATLAS environment are enabled to interact with a subsystem that provides new test capability, and the subsystem can use a test controller based in ATLAS. The work serves as an example that an existing test system can be extended to support new test applications, while demonstrating a client-server based architecture of a subsystem can be extended to support a new operating system in a straightforward way. This paper will describe the client-server extension on the integrated subsystem, the ATLAS Host API executed as FEP on ATLAS, the specifics of communication of composing commands and processing responses, and some examples. This paper will also discuss the considerations behind the architecture and challenges in the development and implementation.

Keywords—ATLAS; integrated subsystem; subsystem communication; client/server; ethernet.

I. INTRODUCTION

The ATLAS Host API project was started to address the lack of a protocol to enable communication between an ATLAS system and an integrated subsystem. The need arises from the enduring legacy of the ATLAS standard, and the introduction of an integrated subsystem. On one side of this divide, ATLAS (Abbreviated Test Language for All Systems), is a legacy test language in the avionics test arena that has been used to write many test requirements and develop test program sets (TPS's), remaining an essential tool for long-term development for upgrades of military and commercial avionics equipment.

Across the aisle is an integrated subsystem, a design developed to work with new developments in avionics technology. As new capabilities require buses with fast data rates, a system comprised of integrated instrumentation and an embedded controlling computer is an approach to address the high throughput requirements of a unit under test (UUT). Along with local test development, the embedded computer also allows for control of the instruments through an external computer, what is named the "test station PC". Actual TPS would run on the test station computer, while a subsystem server component provides an access point that enables the test station PC to start subsystem applications, pass data to and receive data from those applications, manage files on the subsystem PC, report the subsystem configuration, and other tasks.

Existing systems based in ATLAS would like to leverage the extended test functionality that a subsystem provides, and likewise, an existing subsystem realizes the need to extend the test station computer to support an ATLAS system, hence the development of the ATLAS Host API. The following sections of the paper will describe the components of the architecture, and the data block format used in the communication.

II. CLIENT / SERVER ARCHITECTURE

The client/server architecture is a commonly used design to bridge disparate systems. Since ATLAS does not provide language constructs that deal with controlling a subsystem, a FEP-(Functional External Program) based solution is implemented to provide the required control in the test program. The FEP running in the ATLAS system plays the role of the client, issuing requests to and receiving responses from a server program running on the subsystem PC, communicating over Ethernet. The two programs use connection-based network sockets for the communication. The function library for the FEP is referred as the FEP Client API, and serves as the interface to the subsystem.

The program running on the subsystem PC in this setup is called the Remote Host Server. This component receives data over an established connection, and translates the data into commands that are then executed on the subsystem PC.

Together, the FEP Client and the Remote Host Server make up the ATLAS Host API solution.

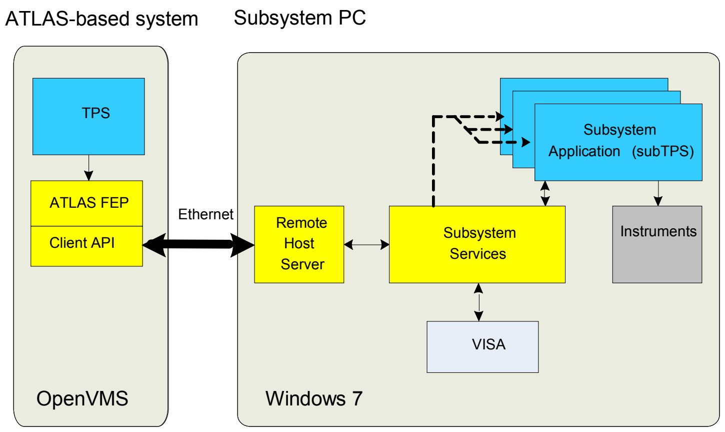

Fig. 1. Software architecture showing access between ATLAS system and the subsystem PC

Fig. 1 shows the overall architecture, with the FEP program running on the ATLAS system making use of the FEP Client API to send commands to the Remote Host Server running on the subsystem PC. The Remote Host Server receives the commands and communicates with the existing subsystem services to perform work such retrieving system configuration and starting test programs.

The Subsystem Application (subTPS) refers to the test program executing on the subsystem PC. It is named "subTPS" due to the similarities between the main test program code (the TPS), as it makes calls to the instrument driver to establish a session with an instrument, issuing setup instructions, initiating measurements or tests, processing the results, and terminating the session. It is natural to think of the instrumentation computer component as being a TPS in its own right, one that works under the direction and control of a TPS [1].

Following are more details of each component.

A. Remote Host Server

The Remote Host Server provides the following services to an ATLAS-based TPS:

- Start subTPS execution
- Report subTPS execution status
- Abort subTPS execution
- Send messages to and receive messages from executing subTPSs
- Reset the subsystem hardware
- Perform subsystem self-test functions
- Report the subsystem hardware configuration
- Send and receive files
- Logging support

These services are initiated by a command sent to the Remote Host Server. The Remote Host Server is a Windows Service process that is started at boot time, and runs in the background. When the server loads, it starts a thread that continuously listens for incoming connection attempts on a pre-defined port number. The thread blocks, or waits, until a client is connected. When a client is connected, the invoked function signals the listener thread to continue listening for the next connection. The function then checks if there is a running data processor thread, and whether this is an existing connected socket. The current design defines only one active connection, so the previous connection is closed, and a running processor thread aborted. This design attempts to make the server more robust, such that any issues that arise in the processing of a data block would not affect the availability of the server.

The function continues next by creating a data processor thread, while setting up a storage object to receive the incoming data stream. The thread starts out in a blocked state, to wait for the signal from the function. The server is set to be in a receive data mode until the total bytes received equals the expected data length. When the values match, the processor thread is signaled to begin processing the data.

As each part of the data block is parsed, it is checked for validity. When an error is found, the server sends a response with an error code. The developer can refer to a list of defined error codes to determine the cause. The server then interprets and performs the requested command, using the associated parameters from the data block. At the end of execution, the server sends a properly formatted response data block with the results and status information.

B. FEP Client API (Subsystem Interface)

The FEP is composed of two components, the ATLAS interface and the subsystem interface. The ATLAS interface is responsible for extracting function parameters out of the executing ATLAS code and for returning data back to the executing ATLAS code. The subsystem interface is responsible for issuing commands to the subsystem and processing responses to those commands. This paper focuses solely on the subsystem interface portion.

The subsystem interface provides functions to create the data blocks, add the command and parameters, send the command data block, and process the response data block. Table I lists the relevant API functions. Each function returns an integer value to indicate success or failure of the operation. Errors pertaining to execution of commands are reported in response data blocks, which are described in the following section.

TABLE I. SUBSYSTEM API FUNCTIONS

Function Prototypes (return types and prefixes omitted)	**Description**
Init (char *ipAddress, int classifiedMode)	Performs any necessary initialization steps.
Reset (float gracePeriod)	Sends a reset command to the Remote Host Server. After reset:

	• All subTPSs will have been terminated • The instrument FW will have been unloaded • The instrument memories will have been cleared
StartCommand (int commandCode, int id)	Creates a command data block.
AddParameter (int paramType, int paramID, void *paramValuePtr);	Appends a simple parameter to the current command data block. Examples of paramType are TERFEP_CHAR, TERFEP_INT, TERFEP_DOUBLE.
AddArrayParameter (int paramType, int paramID, int arraySize, void *arrayPtr);	Appends an array parameter to the current command data block.
SendCommand ();	Completes the creation of a command data block and sends it to the subsystem.
CancelCommand ();	Terminates the current command data block. If the cancel command is received prior to the command being sent to the subsystem, the command is discarded. If the cancel command is received prior to the complete command being sent to the subsystem, the command is terminated. If the cancel command is received after a complete command is sent to the subsystem, then the cancel has no effect.
ReceiveData (double timeoutInSeconds, int *blockTypePtr, int *codePtr, int *idPtr, int *hasParamsPtr);	Waits for an incoming data block to be received from the subsystem. The expected usage and related functions is to wait for the next data block, check the hasParamsPtr, and if true, • call the ExtractParameterInfo function to determine the type and ID of the parameter • call the ExtractParameterValue function to get the parameter value and to determine if there are any more parameters. If so, repeat these steps.
ExtractParameterInfo (int *paramTypePtr, int *paramIDPtr, int *paramCountPtr);	Returns the type information on the next parameter in the current received data block.
ExtractParameterValue (int valueBufferSize, void *valueBufferPtr, int *hasParamsPtr);	Returns the value of the next parameter in the current received data block.
ErrorMessage (int errorCode, int bufSize, char *bufPtr);	Returns a description of an error code.
Close ();	Shuts down communication with the subsystem and frees any allocated local resources.

III. DATA BLOCK FORMAT

Data that is sent over the socket connection between the ATLAS FEP client and the Remote Host Server is specially formatted. There are a few types of data, but they all share the same common structure. The data is organized into variable- length byte arrays, with the general structure listed in Table II:

TABLE II. DATA BLOCK STRUCTURE

Contents	Notes
<header-0> <header-1> <header-2>	Forms the header of a data block and consists of a sequence of characters that is customized for support of customer needs.
<type>	Specifies the type of data block. There are 3 supported types: • Command, indicated by the value 'C' • Response, indicated by the value 'R' • Message, indicated by the value 'M'
<code>	Meaning depends on the data block. Usage is described in the data block section.
<id-0> <id-1> <id-2> <id-3>	These 4 bytes form a 32-bit value that either uniquely identifies the data block (for commands) or specifies the data block to which it applies (for responses).
<param-type>	Denotes the start of a parameter value. It is bit-encoded to indicate information about the parameter, such as size, type, and whether it is an array. Parameters are optional.
<param-id>	Uniquely identifies the parameter for this command.
<param-length-0> <param-length-1> <param-length-2> <param-length-3>	Indicates the length of an array-based parameter. Depending on the length of the array, not all of these bytes may be present (the <param-type> field determines this). These bytes are not present if the parameter represents a simple value.
<param-data-0> … <param-data-N-1>	The value of the parameter. The number of bytes used depends on the data type of the parameter as indicated in the <param-type>.
…	Any number of parameters can be included in the block as well.
<end>	End of the data block.

Data transmission between the two systems is assumed to be in little-endian form, so no byte-re-ordering is needed.

The different data block types will now be described in more detail.

A. Command Data Block

Commands sent over the socket connection from the ATLAS system to the subsystem are packaged as command data blocks. The <type> field for a command data block is the ASCII 'C' character. The <code> field will contain a value denoting the command to execute. Related operations are grouped together, so for example 0x00 to 0x09 is reserved for subsystem operations, 0x10 to 0x19 for self test's, and so on.

Table III shows an example of a command data block calling command 42 (0x2A) that includes 2 parameters: one, a string of characters, "abcdef" with the ID 1, and the second a 32-bit integer with the value 0x12345678 and with ID of 2. This command was given the ID 0x11223344 by the client.

TABLE III. COMMAND EXAMPLE

	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x00	'A'	'A'	'A'	'C'	0x2A	0x44	0x33	0x22
0x08	0x11	0x80	0x01	0x07	'a'	'b'	'c'	'd'
0x10	'e'	'f'	'\0'	0x02	0x02	0x78	0x56	0x34
0x18	0x12	0xFF						

The shaded cells in Table III represent the header fields for the two parameters. The first parameter is a small array of chars, one whose length is small enough to encode in a single byte (0x07). The second parameter is a simple 32-bit signed integer.

A user will not have to labor over the specifics of encodings and implementation, as macros are defined in the FEP client library to assist with data block construction and parsing.

B. *Response Data Block*

Data sent over the socket connection from the subsystem to the ATLAS system in response to commands are packaged as response data blocks. The <type> field for a response data block is the ASCII 'R' character. The <id> field is used to match the response with a particular command. For each command data block, there is exactly one response data block. The response data block may include its own set of return parameter values as well, depending on the command.

The <code> field is used to indicate success or failure of a command operation. If the field is non-zero, then the corresponding command has generated an error. In this case, the response data block contains two parameters: the first is an integer error code, followed by a char array describing the error.

C. *Message Data Block*

Message data blocks can be sent from the subsystem to the ATLAS system as a way to convey debugging or alert messages. The intent behind these data blocks is to provide some level of communication between the Remote Host Server and the FEP process without requiring ATLAS involvement, should the need arise.

The <type> field for a message data block is the ASCII 'M' character. The <id> field is not used, and <code> field can be used, for example, to indicate a severity level for the message. The data block includes a single char array parameter that contains the message.

IV. AN EXAMPLE

Fig. 2 shows a representative FEP code in the ATLAS system to execute a test program (subTPS) on the subsystem.

```
#include "TerFEP.h"
#include "TerFEPerr.h"

int StartTPS(
char *name, char *path, char *args, float ackTime)
{
   StartCommand(CMD_START_SUBTPS, sendBlockID);

   AddArrayParameter(TERFEP_CHAR, 0, strlen(name), name);
   AddArrayParameter(TERFEP_CHAR, 1, strlen(path), path);
   AddArrayParameter(TERFEP_CHAR, 2, strlen(args), args);
   AddParameter(TERFEP_FLOAT, 3, &ackTime);

   SendCommand();

   ReceiveData(
      (double)30.0,     /* timeout in seconds */
      &blockType,       /* == TERFEP_RESPONSE */
      &code,            /* == SUCCESS */
      &recvBlockID,     /* == sendBlockID */
      &hasData          /* == 0 */
   );
}

#define CMD_START_SUBTPS 0x20
...
#define TERFEP_RESPONSE 0x52   /* 'R' */
...
#define TERFEP_CHAR 0x00
#define TERFEP_FLOAT 0x12
...
#define SUCCESS 0
```

Fig. 2. Sample code to send the start subTPS command to subsystem

The code references two header files; the *TerFEP.h* contains the functions of the client API, and *TerFEPerr.h* lists the status codes that are shared between the client and the server. In the StartTPS function the first section sets up a command data block, then sends it to the Remote Host Server on the subsystem. The ackTime parameter is the start-up acknowledgement time used to detect whether a subTPS has successfully started. A typical value is 0, which means to wait indefinitely for the notification from the subTPS.

The final library function call gets the server response for the command. Since the result of this particular command contains no payload, no further processing is needed. The code comments indicate the expected values in this case.

V. CHALLENGES / FUTURE WORK

The major issues faced during the implementation came from unfamiliarity with OpenVMS and VAX, DEC Alpha systems, and the need for a robust server. It was late in the development when it was discovered that older VAX, DEC Alpha machines used a different representation for storing float/double values. On the server front, dropped connections are so commonplace during debugging that procedures were needed to define steps to make the server more robust to disruptions. Having to restart the server every time a communication is abruptly cut off became rather annoying.

There are a couple areas where the Remote Host Server's capabilities can be extended. Currently the server accepts only one connection, meaning it can process only the request from a single client. To support the possibility of an environment with multiple clients, the server needs to allow for multiple connections. A second area concerns the self test component; it is currently implemented as an integrated part of the server. It would be more flexible if the server can at runtime, detect the availability of the self test executable, and load it. Extending this further, the server can check whether all requisites are met for a service, then enable or disable accordingly and notify the client of relevant information.

VI. CONCLUSION

The need to enable communication between an ATLAS system and a subsystem providing instrumentation using high speed buses prompted the development of the ATLAS Host interface library. This paper described the specification, implementation, and use of this library and the accompanying Remote Host Server. TPS developers in an ATLAS environment now have the means to extend their existing test programs by connecting to and make use of an integrated subsystem.

ACKNOWLEDGMENT

The author would like to thank Mike McGoldrick for guidance in the design and development of the API and components, and Stephan Krach for contributions in library development and testing, and as a resource for ideas exchange.

REFERENCES

[1] McGoldrick, M., "The Impact of Test Instrumentation with Distributed Processing Capabilities on Test Program Set (TPS) Architecture and Development", *Proc. AUTOTESTCON*, Baltimore, MD 2011.