



Swell Software, Inc.
3081 Commerce Avenue, Suite 400
Fort Gratiot, MI 48059
www.swellsoftware.com
phone: (810)-385-2893
fax: (810)-385-2947

Technical Overview Documentation for PEG Remote Screen Server and PEG Remote Screen Client

Copyright (c) 2001-2009, Swell Software, Inc.

This document may not be reproduced in any form, in whole or in part, without the express written consent of Swell Software, Inc.

PEG is a registered trademark of Swell Software, Inc.

X Window System, X11 and X11R6 are registered trademarks of the Massachusetts Institute of Technology.

Windows, Windows NT and Win32 are registered trademarks of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

INTEGRITY is a trademark of Green Hills Software.

Other trademarks are property of their respective owners.

Document revision number: 1.0.0

Document number: 200104

Overview

What is it?

The PRESS (PEG Remote Screen Server), is a comprehensive client/server environment for simultaneous execution of separately linked PEG applications which occupy discrete virtual memory regions.

PRESS is a small application built from the core of the PEG library with the addition of a communications layer that allows for PEG Remote Clients (Clients) to remotely interact with the video hardware. PRESS executes on a host system that is physically connected to the video hardware. Client applications may reside on the same host as PRESS, on a disparate host across a network, or both. To PRESS, a Client application residing on the same host is treated exactly the same as a Client application running on a system in the next cubicle, or the next building, or the next country.

PRESS has two basic jobs in relation to the Client applications. The first is to provide display output. It does this by interpreting drawing commands sent to it by the Client applications into terms that the video hardware understands. To do this, PRESS uses the screen drivers provided in the PEG library. Therefore, all of the video hardware that the PEG library supports is automatically supported by PRESS. The second job of PRESS is to provide user input to the appropriate Client application. It does this much the same as the drawing service, but in reverse. Instead of receiving a command from an application and sending it down to the hardware, PRESS interacts with the input hardware, translates that message into a format suitable for the connection to the Client application, and sends it along. When the Client application receives the message, it is pushed into a PEG message queue just as if the hardware interaction had happened within the context of the Client application. By abstracting the Client application away from the hardware, we are able to execute any number of Client applications in conjunction with a single PRESS and bypass the memory restriction issues.

Why is it?

To understand the necessity for such a client/server execution model within the PEG framework, we need to briefly review the basics of how PEG works.

The interface to the display hardware is encapsulated within the PegScreen class. Any derivative of this class is generally referred to as a screen driver and is intended for use in conjunction with a particular piece of video hardware. Every object within a PEG application draws itself by interacting with the PegScreen class. In turn, the PegScreen class interacts directly with the video hardware to rasterize the visual output of the calling object.

Working with the PegScreen class is the PegPresentationManager class. This class provides the windowing framework of the GUI by orchestrating the top level windows of the application. PegPresentationManager also provides the deterministic user input services to the application. In other words PegPresentationManager, while not directly interacting with the user input hardware, determines which top level window should receive input messages from the hardware.

When running a PEG application with a PEG library configured to run as Stand Alone, usually there is no underlying support to run more than a single PEG application at any given time. For instance, when running PEG with DOS, DOS is restricted by its inability to run more than one application at a time.

Micro-kernel RTOS's, which layout system memory in a contiguous byte array, allow the single instance of the PegScreen and PegPresentationManager classes to be shared among any number of executing processes, or tasks. This fits in with the Multi-Threaded Execution Model supported by PEG, where any task can interact directly with the GUI at any time. In this case, the PegPresentationManager provides the decision making services to correctly push user input to the appropriate task.

The Multi-Threaded Execution Model also has a use when executing on an operating system that enforces a protected memory space design. In the case of executing on top of a Unix style operating systems, PegScreen and PegPresentationManager may be shared among threads running within a single process, but not between processes themselves. To a certain extent, this is restrictive in that it forces the application designers to create a single, potentially large, executable that encompasses the entire functionality of their system, instead of allowing them the flexibility of creating any number of small applications that can be dynamically plugged into the system at any given time.

To alleviate this problem, PRESS implements a framework that abstracts the Client applications away from the video hardware and input devices by providing PegScreen and PegPresentationManager services through a

communication protocol that is able to interact between separate processes. In this fashion, we are able to restrict the PegScreen and PegPresentationManager to a single instance on any given piece of hardware without, in turn, restricting the number of PEG applications which may also execute on the same hardware.

An added bonus to this framework is the ability to not only communicate between processes on the same host machine, but to also communicate between processes running on separate machines across a network. This allows for any number of remote Client applications to have their output displayed at a central location.

Of course there are performance trade offs that must be considered with using this execution model such as network latency and protocol overhead, and we will examine this in greater depth in the following sections.

Supported Operating Systems

For a number of reasons, PRESS was originally written on a Unix style operating system, Linux. It has been successfully run and tested on several flavors of Linux (see the table below) with no modification. It also runs as a native FreeBSD application when recompiled on a FreeBSD machine.

Because the communications layer is abstracted away from the protocol spoken between PRESS and any Client applications, PRESS is as portable as PEG itself. In other words, PRESS will run on any of the operating systems that PEG supports providing the operating system is able to implement, either natively or through the use of a standard communications layer such as TCP/IP, a transport mechanism that will allow PRESS to interact with Client applications.

Operating System	Version
Linux	RedHat 6.x LynuxWorks BlueCat Caldera OpenLinux 2.x Slackware 7.1 Lineo Embedix Debian 2.2
FreeBSD	4.x
INTEGRITY	2.0.5
LynxOS	3.0.1

Table 1.
Currently Supported Operating Systems

Interprocess Communications Facilities

To allow for PRESS to communicate to the Client applications across process boundaries, we need some form of Interprocess Communication (IPC) facilities. As delivered PRESS supports a number of IPC implementations to allow for maximum flexibility as well as keeping an eye on performance. In other words, you wouldn't want to use INET domain BSD Sockets if your target OS is INTEGRITY because of the performance penalty. The architecture of PRESS does not lock you into a single IPC option so that you can choose how your Client applications will interact with PRESS in the most efficient way possible while still meeting the requirements of your system.

This communications layer is what allows memory protected operating systems to run multiple Client applications. Without this, a single PEG application running on these operating systems must encompass the entirety of the application in either the Stand Alone or Multi-Threaded execution models as a single binary program.

Since this communication layer is at the core of PRESS concept, we'll take a look at the supported transport mechanisms and discuss their relative strengths and weaknesses, as well as how PRESS abstracts this mechanism to provide a reasonable degree of portability. The following table lists these implementations.

In addition to the communication methods described below, it is possible to substitute your own communication facility such as serial, backplane, or other. This would require some work on the part of the application developer as only those standard IPC types described below are supported with pre-defined build switches.

IPC Mechanism	Supported Operating System	Domain
BSD Sockets using TCP/IP	All Operating Systems listed in Table 1. Will also support any OS that provides socket and TCP/IP facilities.	INET - Allows for local and remote connections.
BSD Sockets using TCP/IP	All Operating Systems listed in Table 1. Will also support any OS that provides socket and TCP/IP facilities.	UNIX - Allows for local connections.
Unix Named Pipes INTEGRITY Connections	Linux, FreeBSD, LynxOS INTEGRITY	Allows for local connections. Allows for local connections. May also provide for remote connections.

Table 2.
Supported Communication Mechanisms

BSD Sockets

First is the BSD Socket form of IPC. When coupled with the TCP/IP protocol, this transport layer forms the backbone of the Internet as well as the majority of network systems in existence. For purposes of our discussion, we'll take a look at the two common domains associated with this protocol, INET and UNIX.

When we refer to the INET domain, as the nomenclature may imply, we are usually referring to a network made up of any number of host machines connected by some physical means. When this domain is specified at build time for PRESS, it uses the socket facilities in a fashion that permits PRESS to accept connections from Client applications that may be running on the same host as PRESS, or that may be running on a host system somewhere in the far corners of an uncharted network. Running in this mode, PRESS is unconcerned with the locality of it's clients. It treats each client as if they were running on a disparate machine in some remote location.

The clear advantage to this model is that PRESS and Client applications run entirely independent of each other. Their hardware platforms may be a mix and match variety. A Client application may be inserted into the execution stream at any time from any host that has a wire connection to PRESS. By the same token, a client application may likewise be extracted without any adverse side effects to PRESS.

The disadvantage of this model lies in the network transport mechanisms. PRESS and it's Clients are at the mercy of network bandwidth. Thus, any latency or bottleneck in the network will effect performance. And while PRESS and Client applications may be executing on the same host which will isolate them from these issues, there is still overhead involved with packet transference between processes on the same host just by virtue of the reliability of the TCP/IP protocol.

A step over from this model is moving the execution out of the INET domain and into the UNIX domain. Since this is still within the TCP/IP protocol, much of the internal functionality of PRESS remains the same. However since we are assuming that all of the clients will be executing on the same host, we make up for a fair portion of the overhead associated with the communication layer.

A downside to the UNIX domain is that PRESS and all of it's clients must execute on the same host. If your particular system involves the need to execute any number of separately linked Client binaries, then this is a more efficient choice since we retain the option of starting up or shutting down any Client client at any time as with the INET model.

A second requirement to consider is that instead of using a "well known" port for connecting the clients to the server, UNIX domain sockets usually will create a node on the file system that acts as a buffer for the data stream being passed back and forth. If your system has a writable file system attached to it, as most desktop style operating systems do, then this would not be an issue.

Overall, there is less overhead in the internal message packaging since we don't have to take byte ordering or structure packing into account, so that adds to the efficiency of this model.

Named Pipes

Named pipes are predominately a Unix IPC phenomenon. This model operates much in the same way as the UNIX domain sockets without incurring the overhead associated with a TCP/IP stack. Therefore, it shares many of the benefits of UNIX domain sockets, but executes faster. The drawback of this design, like the UNIX domain sockets, is that name pipes usually create a node in the file system to use as a buffer for the data stream. This would only be a problem if your target hardware did not have a writable file system available to it.

INTEGRITY Connections

The final communication implementation is specific to Green Hills Software's INTEGRITY real-time operating system. Although INTEGRITY does optionally provide a BSD style socket library and TCP/IP stack, the preferred method for IPC is through the use of Connections. The Connections are implemented in the kernel libraries and since they are under control by the kernel, they do not pose a threat to the stability of the system because the INTEGRITY kernel enforces a very stringent memory protection policy. This model most closely resembles the Unix named pipes model discussed above, except that it will usually use system memory to allocate the communications buffer, whereas Unix named pipes usually create a node in the file system.

The down side to Connections is that they are implemented only on INTEGRITY.

Considerations

Since we have taken great care to isolate the applications from the transport layer, switching between the communication methods defined above requires only that you set a #define in the pconfig.hpp configuration file. You can easily test different models as you develop your application without worrying about what layer your code is using. This will give you a good feel as to the performance of the application running on different communication models. If at any time your requirements change you can easily switch to a model that implements the features that you need.

In summary, the reality of the client/server model is that you will pay a performance penalty for not using the Stand Alone or Multi-Threaded execution models. There is no way around this. There isn't a communication protocol in existence that is faster than an application directly accessing hardware. The PRESS and Client application execution model should only be used in those environments where multiple independantly linked Client application programs are required.

Implementation

New Files

As you would expect from added functionality to the standard PEG library, there are additional source files that are not generally referred to in the PEG Programming and Reference Manual. For the sake of simplicity, all of the new source files are in the same directories as the standard PEG source files. The new header files are in the \$INSTALLDIR/peg/include directory, while the implementation files can be found in \$INSTALLDIR/peg/source.

Table 3 summarizes the names and locations of the new files. The "Included in Which Build" column refers to if this file is included in building the server (PRESS) and/or the client (Client).

File Name	Included in Which Build	Contains New Class(es)
\$INSTALLDIR/include/pressdef.hpp	PRESS, Client	Various defines and communication protocol structures
\$INSTALLDIR/include/press.hpp	PRESS	PegRemoteScreenServer, PRESSWindow, PRESSResourceRegistry
\$INSTALLDIR/source/press.cpp		PRESSMessage
\$INSTALLDIR/include/pressmsg.hpp	PRESS, Client	
\$INSTALLDIR/source/pressmsg.cpp		
\$INSTALLDIR/include/Clientscr.hpp	Client	ClientScreen
\$INSTALLDIR/source/Clientscr.cpp		
\$INSTALLDIR/include/Clientpm.hpp	Client	ClientPresentationManager
\$INSTALLDIR/source/Clientpm.cpp		

<code>\$INSTALLDIR/source/pressini.cpp</code>	PRESS	PegAppInitialize function
---	-------	---------------------------

Table 3.

New Files in the PEG source tree

There are also new make files. These files can be found off of the `$INSTALLDIR/build/press` and `$INSTALLDIR/build/Client` directories for the server and the client library, respectively. Within these directories are subdirectories that are structured like the main PEG build directory. As an example, if you would like to build PRESS to run on Linux, you would go to the `$INSTALLDIR/build/press/linux` directory and run make.