SourceServer v. 3.4.2 Reference

Contents

Introduction

SourceServer is a standalone version of the MPW Shell's Projector that provides users with source code control for a software project. SourceServer performs as a server application that has no user interface; all of its commands come to it via AppleEvents. It is up to client applications to determine which Projector commands they support, what user interface they provide, and to issue the corresponding commands to SourceServer.

Except for features relating to the user interface, we emphasize that SourceServer 3.4.2 is identical to Projector as found in MPW 3.4.2.

Hardware and Software Requirements

SourceServer requires a Macintosh running System 7.0 or later with AppleEvents. SourceServer itself comes preconfigured to use the same amount of memory as ToolServer, which is 768K of RAM. Best performance is achieved if both the cooperating development environment and SourceServer can be co-resident in memory. Using ToolServer as an example client, the total recommended memory requirement is 4 megabytes.

General Description

As was stated in the introduction, SourceServer is a standalone version of the MPW Shell's Projector that provides a means to control and account for changes to all the files associated with a software project. All of the project's files are stored, along with compacted prior versions and history, in a database named ProjectorDB. Subprojects are created by nesting ProjectorDB files within a hierarchy of folders. This nesting is done automatically during the `NewProject` operation. SourceServer is a single application with the creator signature 'MPSP' to differentiate it from the Shell's 'MPS.' However, it performs all Projector functions on ProjectorDB files that have creator signatures of 'MPS.' This is done to allow all projects to be accessed from either the MPW Shell or SourceServer, since Projector actually checks for that signature.

SourceServer performs almost all Projector commands. It also performs the `Directory` command; this command has been included because it is commonly used in Projector scripts. SourceServer is not able to execute scripts; therefore `CompareRevisions` and `MergeBranch` are not available. With respect to the ToolServer/SourceServer combination, there is a further limitation in that those two scripts require an editor to function properly. `OrphanFiles` and `TransferCkid`, which are scripts in the current MPW world, have been transformed into built-in SourceServer commands. Since the "NewProject," "Check In," and "Check Out" dialog windows are not available, the `-w` and `-close` window options of the `NewProject, CheckIn`, and `CheckOut` commands are illegal. A detailed list of commands and supported options is included below in the section titled: Supported Commands.

Each instruction sent to SourceServer must correspond to a command line containing only a single, atomic, Projector command. Since there are no Shell variables or aliases that can be expanded, options should be listed explicitly and pathnames are required to be fully expanded when passed on to SourceServer. There is some leeway in the requirement for fully expanded pathnames: SourceServer maintains a variable, the "current directory," which is the default location from which commands find their targets. Existing Projector scripts use knowledge of the current directory in their execution, which necessitates the inclusion of `Directory` in the list of commands understood by SourceServer. Keeping the current directory setting in synch between SourceServer and the client application allows the usage of partial pathnames. However, since existing MPW scripts assume that Projector uses the same directory variable as other Shell components, the scripts may have no initial `Directory` calls that would synchronize the directories. For this reason, fully qualified pathnames are still the recommended way to specify a target.

The method of sending instructions to SourceServer is the creation of a special AppleEvent mechanism. The client application must communicate with SourceServer by issuing an AppleEvent with a message ID of `'cmnd'`. The exact structure of this AppleEvent is described later in the section titled: Client Development Information.

AppleEvents also provide the mechanism to get replies back from SourceServer. As described later, the reply contains separate sections for return to `stdout` and `stderr`. Actual redirection of output is the responsibility of the client application. Thus, in the equivalent of the command line `ProjectInfo -r > tempFile`, the "`> tempFile`" should be handled locally and not sent on to SourceServer. There is also a limit of 64K of reply imposed by the EPPC transport mechanism used by AppleEvents. If output would be greater than 64000 characters, SourceServer limits the combination of error messages and text output to 64000 characters and returns the error `$41, kAEReplyTooBig.`

SourceServer operates synchronously. It receives the AppleEvent from the client, executes the requested command, and then replies via the reply field of the original AppleEvent.

Although one of the purposes of Projector is to allow access to a Projector database by multiple users, Projector proper has always had a single-user orientation. True to its Projector heritage, SourceServer has no ability to differentiate one user from another. Thus, allowing a single SourceServer to be accessed by multiple users could result in unexpected changing of the current directory or unexpected changing of the current project. Until a mechanism for preserving the state for each user or for providing a lock for each transaction cycle is added to SourceServer, it is not a true multi-user application. SourceServer protects itself from these problems at present by handling only local AppleEvents. This means that SourceServer cannot be operated directly on a remote computer. Variables, such as the current check out directory and mounted projects, are preserved by the local copy of SourceServer. Raw data, represented by ProjectorDB files, can nevertheless be accessed remotely and mounted by several different users at the same time, each using his own Projector or SourceServer.

Human Interface

SourceServer was designed as a faceless server application with all commands being received via AppleEvents. Although no commands can be entered from them, SourceServer does have a status window and menu bar.

The menu bar items provided are the Apple, File, and Edit menus. The Apple menu of course contains "About SourceServer..." in addition to the traditional desk accessories. The Edit menu is normally dimmed, but is available for use by desk accessories and dialogs. The File menu contains the Open Status, Close Status, and Quit items. The Open and Close Status commands control the display of the status window titled Source Status. The Open Status, Close Status, and Quit items are accessed by their keyboard equivalents: command-O, command-W, and command-Q. Additionally, while Projector commands are executing in SourceServer and it is the frontmost application, command period aborts the operation and SourceServer returns the "tool aborted" reply of -9.

While SourceServer is in operation, the name of the currently operating command is displayed in the “Source Status” window if it is open. The state of the window, whether it is open or closed and its location are stored in a preferences file: “System Folder: Preferences: MPW: SourceServer Prefs”. If the file or folder does not exist on startup of SourceServer, it creates them. This is also the folder in which ToolServer stores its preferences.

Another file created by SourceServer is SourceServer.Log. It is created in the same directory as SourceServer and is used to store output only if the output cannot be returned by the AppleEvent reply.

Combining with ToolServer

As has been previously noted, SourceServer is a single-user server that cannot be used on a remote machine. On the other hand, ToolServer, which does run remotely, cannot perform Projector operations. Combining these two applications enables the capability of performing Projector operations remotely. To use this capability, we include a tool to relay Projector commands to SourceServer and some setup scripts. Relaying Projector commands from ToolServer to SourceServer allows the client application to behave as if it were asynchronous, since ToolServer buffers the replies and later sends them back to the client. Remote Projector operations are also performed by this combination of applications as long as SourceServer is on the same machine as ToolServer, and is running before any target scripts containing Projector commands are executed.

The key to binding the two applications together is a tool: `RProj`. `RProj` accepts all the parameters passed to it by the Shell/ToolServer, packages them up into a 'cmnd' AppleEvent, sends the event off to a locally running copy of SourceServer, and finally prints the reply to stdout and stderr. If what follows the RProj command is a valid Projector command line, it is passed on to SourceServer and executed there. Scripts constructed specially to run on a ToolServer/SourceServer combination could contain lines such as:

```
RProj MountProject "{MPW}Examples:Projector
      Examples:Sample:"
```

However, to run existing scripts or those which must be able to run in both an MPW Shell Projector environment and with ToolServer and SourceServer, we must be a little more creative. We can first alias the Projector command to be `RProj <projectorCommand>`. Then when the regular command line is encountered, it will be forwarded by RProj to SourceServer:

```
Alias MountProject "RProj MountProject"
MountProject "{MPW}Examples:Projector Examples:Sample:"
```

Rather than requiring that each script to be executed contains a full list of Projector aliases, two scripts with a master list of aliases, `AliasSourceServer` and `UnaliasSourceServer`, are provided to do this. Notice that simply commanding `AliasSourceServer` is not sufficient. We must actually say `Execute AliasSourceServer` in order to give the aliases a scope beyond the execution lifetime of the `AliasSourceServer` script itself. One further problem must be surmounted before existing scripts run without modification by ToolServer and SourceServer. The current directories of both applications must be synchronized. The `Directory2` script is provided to change both the ToolServer and SourceServer current directories simultaneously. Furthermore, `AliasSourceServer` also contains the alias of `Directory` to `Directory2`. It calls `Directory2` with no arguments to discover where ToolServer is focusing and then sets SourceServer to the same directory. As a result, the Projector commands performed by SourceServer and non-Projector commands performed by ToolServer operate on the same default target files. Following this, it is possible to issue any list of Projector commands or scripts containing Projector commands and have all Projector requests routed to SourceServer with no further modification. Finally, execution of the `UnaliasSourceServer` script restores ToolServer to normal operation.

Example:

```
Execute AliasSourceServer
MountProject "{MPW}Examples:Projector Examples:Sample:"
Checkout -project Sample -d "{MPW}" "command pages"
Execute UnaliasSourceServer
```

Supported Commands

The following section addresses all the commands available in Projector and how to invoke them from SourceServer. All commands are sent as a list of text items in a 'cmnd' AppleEvent as described in the AppleEvents section of this Release Note. `CompareRevisions` and `MergeBranch` are scripts which require an editor to display and confirm their actions interactively, so they are beyond the scope of SourceServer. Since scripts cannot be operated from SourceServer, `OrphanFiles` and `TransferCkid` are built-in to be available from SourceServer. All remaining Projector commands, including the new `CheckOut -history`, `ProjRename`, `ObsoleteProjectorFile`, and `UnobsoleteProjectorFile` are available.

CheckIn

`CheckIn` works exactly as defined in MPW Projector documentation with the exception of the `-w` and `-close` options. Since there are no windows available from SourceServer, these options cause an error message.

CheckOut

CheckOut works exactly as defined in MPW Projector documentation with the exception of the `-w` and `-close` options. Since there are no windows available from SourceServer, these options cause an error message.

CheckOutDir

`CheckOutDir` works exactly as defined in MPW Projector documentation. Remember that directories must be specified by their full pathnames.

CompareRevisions

CompareRevisions is an MPW script that currently produces a difference file containing MPW scripting language needed to navigate the MPW editor. SourceServer can neither interpret the script nor display the text in edit windows. CompareRevisions is therefore not available in this version of SourceServer.

DeleteNames

`DeleteNames` works exactly as defined in MPW Projector documentation.

DeleteRevisions

DeleteRevisions works exactly as defined in MPW Projector documentation.

Directory

`Directory` is not normally listed as a Projector command. However, SourceServer does retain the MPW Shell notion of a current directory. To set that directory, this command is also available in SourceServer. It works exactly as defined in MPW Shell documentation.

MergeBranch

`MergeBranch` is an MPW script that requires an editor to display the differences between two versions of a file and to allow interactive confirmation as they are merged into a single, unified one. SourceServer can neither interpret the script, nor display the text. Therefore, MergeBranch is not available.

ModifyReadOnly

`ModifyReadOnly` works exactly as defined in MPW Projector documentation.

MountProject

`MountProject` works exactly as defined in MPW Projector documentation.

NameRevisions

`NameRevisions` works exactly as defined in MPW Projector documentation.

NewProject

`NewProject` works exactly as defined in MPW Projector documentation with the exception of the `-w` and `-close` options. Since there are no windows available from SourceServer, these options cause an error message.

ObsoleteProjectorFile

`ObsoleteProjectorFile` works exactly as defined in MPW Projector documentation.

OrphanFiles

`OrphanFiles` is now a built-in command of SourceServer and works exactly as defined in MPW Projector documentation.

Project

`Project` works exactly as defined in MPW Projector documentation.

ProjectInfo

`ProjectInfo` works exactly as defined in MPW Projector documentation. There is one additional note, however: Since `ProjectInfo` is the Projector command which is capable of outputting the most text, it is the command most likely to return a `kAEReplyTooBig` error.

RenameProjectorFile

`RenameProjectorFile` works exactly as defined in MPW Projector documentation.

TransferCkid

`TransferCkid` is now a built-in command of SourceServer and works exactly as defined in MPW Projector documentation.

UnmountProject

`UnmountProject` works exactly as defined in MPW Projector documentation.

UnobsoleteProjectorFile

`UnobsoleteProjectorFile` works exactly as defined in MPW Projector documentation.

Client Development Information

The following section contains information on the AppleEvents used to communicate with SourceServer. It is necessary reading only for developers who are creating client applications that control SourceServer to provide source control features.

Core AppleEvents

SourceServer supports the required AppleEvents: `Open Application`, `Open Documents`, `Print documents`, and `Quit`, which are usually sent by the Finder. `Open Application` performs a normal startup. No further action is taken until another AppleEvent is received or the user selects a menu option. `Open Documents` is in response to someone double-clicking a ProjectorDB file that had a creator type of 'MPSP'. At this time, new projects are created with a creator type of 'MPS ' for compatibility with projects created by the MPW Shell. Since there are normally no documents to open, the `Open Documents` AppleEvent is responded to as if an `Open Application` AppleEvent had been received. `Quit` works normally. `Print` is a null operation.

‘cmnd’ AppleEvent

There is one further AppleEvent, specific to SourceServer, that serves to pass Projector commands to it. SourceServer communicates with only this AppleEvent of type `'MPSP'` and message ID `'cmnd'`. The main part of the event message is a list of text that

corresponds directly to a single Projector command line. Since there is no sharing of the MPW environment variables, all pathnames must be fully expanded when they are passed to SourceServer. These commands are parsed directly by the current Projector command interpreter. It uses the count of the items in the list as a standard argc and the list itself as argv. The event message is summarized below:

Message class:	`'MPSP'`
Message ID:	`'cmnd'`
Parameter Keyword:	`'----'`
Parameter Type:	`'TEXT'`
Parameter Data:	string containing fully expanded Projector command line to execute

The event reply that is returned is also of type text, but has more parts:

`'----'`	output as text
`'diag'`	error or diagnostic text output
`'errn'`	system OSErr
`'stat'`	MPW status

Source Code Example—SendCmnd Tool

This is meant to serve as an example to third party implementors who would like to send AppleEvent commands to SourceServer.

```
/*------------------------------------------------------------------------
NAME
      SendCmnd -- a tool to send command lines to SourceServer via AppleEvents

COPYRIGHT
      Copyright Apple Computer, Inc. 1991-1992
         All rights reserved.
------------------------------------------------------------------------*/

long main(int argc, char *argv[])
{
   AppleEvent     message;
   AppleEvent     reply;
   AEAddressDesc  targetAddr;
   AEDescList     params;
   OSErr          myOSErr;
   // replyErr == errAEDescNotFound is OK, so separate from myOSErr.
   OSErr          replyErr;
   OSErr          errReturn = noErr;
   long           projStatus = noErr;
   char           *toolName;
   char           *projOutput;

   Size           size, realSize;         // size of return result
   DescType       theType;                // type of return result
```

```
   int               i;

   // We will need a valid A5 when we do the context shift back to here after
   // sending the AppleEvent, but while we are still waiting for a reply.
   // Otherwise our default wait routine will call SystemTask and die!

   InitGraf(&qd.thePort);
   SetFScaleDisable(true); // So InitGraf does not affect stdout or stderr.

   toolName = argv[0];
   if (argc <= 1)
      {
      fprintf (stderr, "### %s error.\n", toolName);
      fprintf (stderr, "# Must follow %s with valid Projector command
                        line.\n", toolName);
      return 50;
      }

//    Construct the AppleEvent:

   myOSErr = AECreateDesc (typeApplSignature, (Ptr) &SAPSig, sizeof (SAPSig),
                           &targetAddr);
   if (!myOSErr)
      {
      myOSErr = AECreateAppleEvent (kSAPEventClass, 'cmnd', &targetAddr,
                     kAutoGenerateReturnID, kAnyTransactionID, &message);
      if (!myOSErr)
         {
         // create AE holder for parameter list

         myOSErr = AECreateList (NULL, 0, false, &params);
         if (!myOSErr)
            {
            // create parameter list for AE record
            // (skip 1st parameter which is sending tool's name.)

            for (i = 1; i < argc; i++)
               {
               // append each argument to the list

               myOSErr = AEPutPtr (&params, 0, typeChar, argv[i],
                                   strlen (argv[i]) + 1);
               if (myOSErr) break;
               }
            if (!myOSErr)
               {
               // move parameter list to AE record

               myOSErr = AEPutParamDesc (&message, keyDirectObject, &params);
               if (!myOSErr)
                  {

                  // Send AppleEvent and wait for reply:

                  myOSErr = AESend (&message, &reply, kAEWaitReply,
                            kAENormalPriority, kAEDefaultTimeout, NULL, NULL);
                  if (!myOSErr)
                     {
```

```
      //    We got a reply! Tear it apart!

// Separate error returned for AppleEvent transmission errors-

      AEGetParamPtr (&reply, keyErrorNumber, typeLongInteger,
            &theType, (Ptr) &errReturn, sizeof (errReturn),
            &size);

// - versus errors completing MPW shell operations.

      AEGetParamPtr (&reply, kMPWStatusEvent, typeLongInteger,
            &theType, (Ptr) &projStatus, sizeof (projStatus),
            &size);

      // Special case if AppleEvent reply was too big to be
      // transported back by the EPPC transport mechanism.

      if (errReturn == kAEReplyTooBig)
         {
         fprintf (stderr, "### %s error.\n", toolName);
         fprintf (stderr, "# Output exceeds AppleEvent reply
                           buffer limits.\n");
         }

// The design of the cmnd AppleEvent was made with separate
// reply fields for stdout and stderr. This was done to support
// redirection of those two types of output. Unless there is an
// explicit flush of the buffers, the runtime libraries will
// normally send a tool's error messages to the worksheet
// before sending stdout. So we send out our replies in the
// same order.

      //    Display the remote tool's error messages:

      // We don't know what size of text we are sending back,
      // so request 0 bytes and see what the real size is.

      replyErr = AEGetParamPtr (&reply, kMPWErrorEvent,
            typeChar, &theType, NULL, 0, &realSize);

      // Then request storage for correct amount and recall
      // AEGetParamPtr to transfer the real text. 2 calls may
      // seem wasteful, but it uses the least amount of
      // storage. This seemed like a major goal when several
      // cooperating apps (SourceServer and client) must share
      // memory at the same time.)

      if (!replyErr && (realSize > 0))
         {
         projOutput = malloc(realSize + 1);
         AEGetParamPtr (&reply, kMPWErrorEvent, typeChar,
               &theType, (Ptr) projOutput, realSize, &size);

         // AEGetParamPtr doesn't null terminate string,
         // so we do it manually.
         projOutput[realSize] = 0;
```

```
                        // Actually display the error message.
                        fprintf(stderr, projOutput);
                        free(projOutput);
                        }

                     //    Display the remote tool's standard output:

                     replyErr = AEGetParamPtr (&reply, keyDirectObject,
                            typeChar, &theType, NULL, 0, &realSize);

                     if (!replyErr && (realSize > 0))
                        {
                        projOutput = malloc(realSize + 1);
                        AEGetParamPtr (&reply, keyDirectObject, typeChar,
                           &theType, (Ptr) projOutput, realSize, &size);
                        projOutput[realSize] = 0;
                        fprintf(stdout, projOutput);
                        free(projOutput);
                        }

                     AEDisposeDesc (&reply);
                     }
                  else if (myOSErr == connectionInvalid)    // from AESend()
                     {
                     fprintf (stderr, "### %s error.\n", toolName);
                     fprintf (stderr, "# Unable to connect with
                                          SourceServer.\n");
                     }
                  }
               }
            AEDisposeDesc (&params);
            }
         AEDisposeDesc (&message);
         }
      AEDisposeDesc (&targetAddr);
      }

   // There is a hierarchy of possible errors; the status returned to the
   // shell is the first non-zero value from these possible problems:
   //
   // 1. Report errors from this tool's calls.
   // 2. Report OS errors from Projector calls in SourceServer.
   // 3. If tool OK, report SourceServer's Projector status.

   if (myOSErr == noErr) myOSErr = errReturn;
   if (myOSErr == noErr) myOSErr = projStatus;
   return myOSErr;
}
```