ScriptCheck
User's Manual

Document Version 1.10 - August 18, 1998

Table of Contents

ScriptCheck Basics

Error Checking

Script Updating

ScriptCheck Basics

This chapter provides an overview a command line summary of the MPW ScriptCheck tool.

Overview

The Installer Script Checker, referred to as 'ScriptCheck' from this point forward, is a tool that provides the script writer with script error and consistency checking and script completion. ScriptCheck can be quite valuable in debugging flaws in Installer scripts. It will inform the scriptwriter of inconsistencies such as missing resources, incorrect formats, and other errors that can cause a script to be unusable. ScriptCheck also completes a script by doing the tedious chores of filling in atom and package sizes and other required and optional fields in an Installer script.

This document describes the features of ScriptCheck and explains how to use the tool. It assumes a prior knowledge of Installer script structure, design, and resource types. For more information on these topics consult the document *Installer Engine Guide.*

ScriptCheck was written to help you debug and finish your Installer scripts. Although it can find many errors and problems in a script, it is a preprocessor and cannot check the run-time robustness of your script. Just as a Pascal compiler may smile on a piece of code that later executes an infinite loop, ScriptCheck may approve of a script that the Installer later barfs on!

To help catch as many bugs in your script as you can, use the Installer Debugger is conjunction with ScriptCheck.

Good luck.

Command Line

ScriptCheck is an MPW tool that is executed from the MPW WorkSheet. Run ScriptCheck **after** you have Rez'd your Installer script and fixed any errors that Rez reported.

After starting ScriptCheck, the only action required by the user involves inserting the source disks. For those scriptwriters that develop their installation scripts utilizing folders on their hard disk in lieu of the actual installation disks, the insertion of source disks is necessary only if ScriptCheck fails to find the source files within the "install disk" folders. The rest of the process is automatic, but can take time depending on the size of the script.

`ScriptCheck ( [-a] [-d] [-f] [-h] [-l0…2] [-p] [-q] [-r] [-s] ) scriptFileName`

Options

All of the command line options, in parenthesis above, are optional.

-a Tells ScriptCheck to update the size fields for ALL 'infa' and 'inra' resources, regardless of whether they were manually entered. This option may be useful when re-checking a script, which is NOT recommended.

-d Tells ScriptCheck to update ALL 'infs' creation dates regardless of the value it finds in the creation date field. This feature is useful when ScriptChecking a script that has previously been checked. If you get a new set of source disks and the only changes are file sizes and creation dates, then you can run ScriptCheck again with the -d option to update those dates.

This feature should be used only in special circumstances; it is much better to start with a fresh Installer script each time you run ScriptCheck.

-f Tells ScriptCheck to update ALL Finder flags regardless of the value it finds in the Finder flags field. Otherwise, only those files that have zero in this field will be updated. This feature is useful when ScriptChecking a script that has previously been checked.

This feature should be used only in special circumstances; it is much better to start with a fresh Installer script each time you run ScriptCheck.

-h Forces ScriptCheck to restrict its search for the source files to the folders surrounding the script file. ScriptCheck uses the same source folder search strategy as Installer Engine. Without this option ScriptCheck automatically asks for a source floppy disk, ignoring source files near the script file.

If you do not have some or all of the source disks available, you can still run ScriptCheck to verify the correctness of your script itself without checking the source disks. This is accomplished by pressing the "Skip Disk" button in the dialog box that requests a source disk. Please note that skipping a disk may cause the size calculation that is placed in the 'inra', 'infa', and 'inpk' to be incorrect (because the file sizes were not extracted) and creation dates will not be filled in for 'infs' resources that reference the skipped disk. The skip disk feature should only be used for preliminary script development, not for final versions of the script.

-l0…2 Overrides the default error checking level. The default level is level 2, which causes ScriptCheck to output all error, warning, and note messages during its check. This is the most stringent checking level and displays any problems, minor and major, that it finds. Level 0, on the other hand, is the most

lenient checking level. ScriptCheck will only output error type messages when set to this level. Only use level 0 when you are confident that you can disregard any warnings and notes that would have otherwise been output on a more stringent level. Finally, level 1 is a moderate checking level that outputs both errors and warnings, but does not output note messages. In some cases, the note messages may be superfluous information, and you may prefer to use level 1 to avoid them.

-p Causes ScriptCheck to output progress information as it proceeds through its checking cycle. This information is useful because it gives a context to the error, warning, and note messages that are output as script problems are encountered. ScriptCheck will tell you where it is in the checking process and this can help you understand the messages that it presents.

-q Tells ScriptCheck to perform a quick check of the script; to not ask for the source floppy disks. This option is used for preliminary development of scripts. The size fields and creation date fields for the various scripts will not be filled in.

-r Causes ScriptCheck to display “dead” script resources. A “dead” resource is a resource that is found in the script file, but is never referenced by any script resources. For example, if an ‘infs’ exists in the script, but is never referenced by a ‘inrl’, ‘infa’, or other script resources that can reference an ‘infs’, then it is labeled a “dead” script resource.

-s Causes ScriptCheck to strip out “dead” script resources. As described above, “dead” script resources are those resources found in the script that are never referenced by any other script resources. This strip option removes these resources, minimizing script size. The next time you Rez the script you could choose to remove these dead resources yourself or you can continue to have ScriptCheck do it for you. Use this feature very conservatively; if a particular resource that ScriptCheck says is “dead” is really not dead, then ignore the note and do NOT use the **-s** option to do the stripping.

Error Checking

This chapter describes the error checking performed by ScriptCheck.

The complexity of Installer scripts increases the probability of errors. The types of errors fall into two categories: consistency errors and logic errors. Consistency errors, which are more easily diagnosed and corrected, include missing information, such as nonexistent file specs that are referenced in a script, and also extraneous information, such as resources that are present in the script but are never referenced. Consistency checking also includes the files, resources and fonts, expected on the source disks. ScriptCheck can guarantee that every file, resource, and font referenced in the script is present on the source disks. ScriptCheck provides detailed error messages explaining exactly what files, resources, or fonts expected on the source disks are missing and also what script resources are missing from a script.

Errors in logic are often more subtle than consistency errors. For example, the flag bits that are used in many Installer resource types easily allow for logic errors. These flag bits allow the script writer to specify that a file is update only, copy data fork, delete on remove, and many other options. Certain combinations of these bits are completely meaningless or, in other cases, just ambiguous. For example, if the update only bit is set and the copy bit is clear then the file will not be copied to the target even if it already exists on the target. This is not completely wrong, but it is useful for ScriptCheck to inform the writer that the update will not occur with the copy bit clear, which alerts the writer to a possible error. Diagnosis of errors in logic also include range checking of values such as target disk size for logic flaws. For example if the minimum disk size is greater than the maximum disk size (and the maximum is not zero, which is a special case), the writer is alerted that this will never be true and the rule will never fire.

ScriptCheck considers some errors more severe than others. A three-level error reporting mechanism provides the scriptwriter with a precise idea of the severity of the problem. The most severe problems are termed **Errors**, followed by less severe **Warnings**, and finally the least severe **Notes** messages. An Error must be fixed for the script to be usable by the Installer. Warnings can be fixed in order to guarantee robustness of the script, but may not cause noticeable problems. Note level messages can often be ignored, but can pinpoint subtle problems. You can easily differentiate between these three types of messages as ScriptCheck prefixes each message with either:

`#Error:`, `#Warning:`, or `#Note`.

Consistency Errors

The checking performed on the script itself assures the writer that all referenced resources exist and also that all resources in a script are necessary. With disk space becoming a major concern at this time, removing unneeded resources from scripts can be an important concern of scriptwriters. The second form of consistency checking is done on the source disks to confirm that all files and resources referenced in the script exist in the designated places on the proper disks.

The checks will guarantee existence of all:

- 'inrl' resources referenced by 'infr' resources
- 'inpk' resources referenced by 'inrl' resources
- 'icmt' resources referenced by 'inpk' resources
- 'infa', 'inra', 'inbb', 'inaa', 'inat', 'inff', and 'inpk' resources referenced by 'inpk' resources
- 'infs' resources referenced by 'infa', 'inra', inbb', 'inat', 'inff', and 'inrl' resources
- files on source disks that are referenced by 'infs' resources
- 'ICON', 'icl4', or 'icl8' resources referenced by 'icmt' resources
- User function code resources referenced by 'inrl' and 'inaa' resources. These user function resources must exist in the script itself.
- 'boot' resources referenced by 'inbb' resources
- 'sfnt', 'NFNT', 'FONT', and all point sizes of font resources referenced by 'inff' resources.

If any referenced resources are not found, then informative error messages are displayed.

For example, if an 'inrl' references an 'infs' and the 'infs' is not found in the script, the following message will be output:

```
# Error:  'infs' ID = 258 referenced from 'inrl' ID = 997 does not exist.
```

The script writer can then immediately go back to the script sources and figure out why 'infs' ID = 258 was not Rez'd into the script.

Similarly, if a file cannot be located on the source disks, a message such as this is displayed:

```
# Error:   Source file Cool Demos:Lotus 123?? referenced from 'infs' ID = 3005 not found on source disk.
```

The scriptwriter can then determine whether the 'infs' specified an incorrect path or the file was simply not in the correct location on the source disks.

The consistency checks also alert the scriptwriter to "dead" resources that are not needed in the script. A "dead" resource is a resource that is found in the script file, but is never referenced by any script resources. For example, if an 'infs' exists in the script, but is never referenced by a 'inrl', 'infa', or other script resources that can reference an 'infs', then it is labeled a "dead" script resource.

The checks will ensure that all:

- 'inrl' resources are actually referenced by the one 'infr' resource
- 'inpk' resources are actually referenced by at least one 'inrl' or 'inpk' resource

- 'infa', 'inra', 'inbb', 'inat', and 'inff' resources are actually referenced by at least one 'inpk' resource
- 'icmt' resources are actually referenced by at least one 'inpk' resource
- 'infs' resources are actually referenced by at least one 'infa', 'inra', inbb', 'inat', 'inff' or 'inrl' resource
- other resource types in the script itself are referenced at least once in the script

ScriptCheck enumerates all extraneous references in the script. You'll need to use the -r option to calculate this information. For example, an unreferenced 'infs' will be highlighted with the following message:

```
# Note:    Script resource 'infs' ID = 257 was never referenced in the script.
```

Multiple file spec ('infs') resources referring to the same file are another common problem seen in Installer scripts. Because of the special way the Installer deals with files, this problem can confuse the Installer. Therefore, it is essential that the scriptwriter remove any duplicated file specs.

Specifically, all source file specs must be unique and only one 'infs' resource should be used to point to each source file used in the script. In addition, all target file specs must be unique and only one 'infs' resource should be used to point to each target file used in the script. ScriptCheck checks your script to make sure that you have followed this important rule. If it finds 'infs' resources that duplicate a path to a source or target file, it warns you of your error. An example output of this problem follows:

```
# Warning: 'infs' resources ID = 3005 & ID = 3002 refer to the same source file ('FileMaker Data:JustAFile'). This can cause the Installer problems, remove the duplicates!
```

After the check ends, it is the script writer's responsibility to remove duplicate 'infs' resources and fix all 'infa', 'inra', and other script resources which reference the duplicates. Change those resources to reference the 'infs' resource that will remain in the script.

Logic Errors

The main type of logic errors that can get into a script result from improper setting of the flag bits used in the file and resource atom resources. There are a great number of permutations of these bits that result in illogical or ambiguous actions. For example, if the *keep existing* and the *update only* bits in a resource atom flag are both set, then this tells the installer that if the file already exists on the target, then keep the existing one, but also update the file if the file exists.

The file will never be updated with these two bits set together and the script writer is alerted to this illogical setting. An example ScriptCheck output of this type of logic errors follows:

Warning: The flag bits in 'inra' ID = 2000 are set copy, keepExisting, and updateOnly. This combination makes the resource atom never update because it will always keep an existing copy.

ScriptCheck also range checks values that can be entered in the 'inrl' script resources.

The 'inrl' values that are checked include:

- checkMinMemory: make sure the value is feasible (0 <= value <= 16MB)

Error Checking

- checkTgtVolSize: make sure the minimum is not greater than the maximum value (aside from the special case when max is 0)

If either of these values is out of reasonable bounds, warning messages alert the scriptwriter of possible problems.

Script Updating

This chapter describes the tasks ScriptCheck performs to help make keeping your script up to date easy.

Script Completion

Some script resource types require information that is very tedious and cumbersome to calculate. For instance, the size field that is found in the 'infa' must contain the actual size (in bytes) of the file on the source disks. Filling in this type of field is tedious and it must be updated each time the files on the source disks change sizes.

Filling in Size Fields

Installer Engine requires certain size fields in atoms have the exact size the file or resource parts will occupy on the target file. ScriptCheck makes sure these fields are set properly by looking at the source file.

ScriptCheck performs the following activities regarding size fields:

- For File, Resource and Font Atoms, ScriptCheck fills in the exact target size for each specified part. If your files are compressed, and you are using an Atom Extender to decompress the file or resource, then you must supply a code resource that ScriptCheck calls for each compressed file to obtain the original file or resource's size information. See "File/Resource Info Extension Reference" later in this section.
- The total size field in the atom is filled using size information of the parts being installed with the atom.
- The size field in all packages are updated with the total of the containing atom total sizes. This allows the Installer to verify that all target disks have sufficient disk space to complete the installation. This size field is also the value shown to the user in the info window for packages that have the ShowOnCustom flag set.

NOTE

ScriptCheck has problems with updating the individual size fields of multiple source files when the data fork contains the resource fork data. Therefore it is best to copy both forks of a file that is split into multiple source files by using two File Atoms, one to copy the data fork, and the other to copy the resource fork. ◆

Filling in Source Version Fields

Installer Engine allows the scriptwriter to determine the newness of the target resource or file relative to the source using a version number. A field exists in the new formats of the Resource and File Atom that hold the version number of the source. ScriptCheck will automatically update this field if the scriptwriter chooses to supply the necessary custom code resource to determine the source version number.

If the Version Compare rsrc ID field is non-zero ScriptCheck looks for an 'scsv' resource, which describes how to call the code resource that will determine the source version number. The version number returned by the code resource will be entered into the source version number field. The interface of this code resource is described in the section "File/Resource Info Extension Reference".

For File Atoms that contain zero in their version compare resource ID field and are using the `useVersProcToCompare` flag, ScriptCheck will automatically enter the version number from the 'vers' ID 1 resource found in the source file. If no 'vers' ID 1 resource is found the field is left untouched.

Filling in Source Creation Dates

In order to ensure that the Installer finds the proper files on the source disks, the creation date of the files must be placed in the each source file spec. If the scriptwriter places a **1** in the creation date field of the 'infs' resource, ScriptCheck will replace that with the creation date that it finds when scanning the source disks. In addition, a command line parameter tells ScriptCheck to update ALL creation dates regardless of whether a 1 exists in that field (see "Using ScriptCheck" section below). To manually fill in the creation date field, simply place a value other than 1 in that field.

Filling in Finder Flags

By default, ScriptCheck will update the Finder flags field in 'infa' resources if the field contains zero. To force all Finder Flag fields to be updated, regardless of their prior value, use the -f option.

Filling in Package Comment Release Dates

The 'icmt' (or 'inpc') package comment resource requires a field that specifies the version release date of the software in that package. This is a LongInt value that corresponds to the number of seconds between January 1, 1904 and the release date. In the past, the scriptwriter had to calculate this value and place it in the 'icmt' resource. ScriptCheck allows a much simpler method to get the correct LongInt value in this resource. Instead of entering a value in seconds in this field, the scriptwriter can set a numeric representation of the date in the form *mmddyyyy* in this field and ScriptCheck will convert it to the correct value in seconds. For example, if the release date is February 5, 1990, then place 2051990 in the date field of the 'icmt' resource. Note: A month value less than 10 does not need to be preceded with a 0, but a day value less than 10 must be preceded with a 0 as is shown in the example. To fill in the release date field manually, place a true seconds LongInt value in that field and ScriptCheck will not modify it.

Updating the Script Creation Date Resource (‘incd’)

ScriptCheck will always automatically create or update the ‘incd’ resource in the script file. This resource is used by the Installer to help find source folders on AppleShare volumes.

Updating the Script Size Resource (‘insz’)

ScriptCheck will always automatically create or update the ‘insz’ resource in the script file. This resource is used by the Installer to calculate the size of the script sub-heap.

File / Resource Info Extension Reference

This section describes the function interface and resource descriptions needed to use the ScriptCheck File / Resource Info Extension script resource. The function prototype and parameter block is defined in file “TargetInfoMgt.h”.

Function Interface

File/Resource Info Extension

ScriptCheck calls your file / resource info extension code resource when the Atom Extender resource ID entered in the atom is non-zero. ScriptCheck looks for an ‘scex’ resource in a file with the name “scriptFileName.scx”, where scriptFileName is the name of the script you are checking. This resource is almost identical to the actual Atom Extender (‘inex’) resource, which describes how to call the code resource containing the file / resource info function.

The file / resource info function must have the following definition and be the entry point of the code resource.

```
typedef OSErr  (*TargetInfoProcPtr)( TargetInfoPBPtr );
```

The result is defined as an `OSErr`. Returning any other result besides `noErr` tells ScriptCheck to use the information from the actual source file found on the source disk and disregard any information passed back through the parameter block.

Version Info Extension

ScriptCheck calls your version compare extension code resource when the version compare resource ID entered in the atom is non-zero. ScriptCheck looks for an ‘scvc resource in a file with the name “scriptFileName.scx”, where scriptFileName is the name of the script you are checking. This resource is almost identical to the actual VersionCompare (‘invc’) resource.

The version info function must have the following definition and be the entry point of the code resource.

```
typedef long   (*TargetVersProcPtr)( TargetVersPBPtr );
```

The result is defined as an `OSErr`. Returning any other result besides `noErr` tells ScriptCheck to use the information from the actual source file found on the source disk and disregards any information passed back through the parameter block.

Parameter Block

File/Resource Info Extension Parameter Block

The file/resource info function code resource is passed a pointer to a parameter block containing fields that you will use to pass back original source information to ScriptCheck.

```
#define  kDataTypeIsFile   -1
#define  kDataTypeIsRsrc   1

typedef struct {
 ->   FSSpec      fSrcFSSpec;
 ->   long        fSrcDataType;
 ->   Str31       fTgtFileName;
<-    short       fTgtFinderAttrs;
<-    long        fTgtDataForkSize;
<-    long        fTgtRsrcForkSize;
<-    long        fTgtCreationDate;
<-    ResType     fTgtFileType;        // added for ScriptCheck 4.0.3
<-    ResType     fTgtFileCreator;     // added for ScriptCheck 4.0.3
<-    long        fTgtModDate;         // added for ScriptCheck 4.0.3
} TgtFileInfoType;

typedef struct {
 ->   FSSpec      fSrcFSSpec;
 ->   long        fSrcDataType;
 ->   OSType      fSrcRsrcType;
 ->   short       fSrcRsrcID;
 ->   OSType      fTgtRsrcType;
 ->   short       fTgtRsrcID;
<-    short       fTgtRsrcAttrs;
<-    long        fTgtRsrcSize;
} TgtRsrcInfoType;

typedef union {
   TgtFileInfoType      fFileInfo;
   TgtRsrcInfoType      fRsrcInfo;
} TargetInfoPB, *TargetInfoPBPtr;
```

Field descriptions

`fSrcFSSpec`	A FSSPec to the source file referenced by the atom that ScriptCheck supplies you.
`fSrcDataType`	ScriptCheck supplies one of two values: `kDataTypeIsFile` if ScriptCheck wants information about a file; otherwise, `kDataTypeIsRsrc` if ScriptCheck wants information about a resource, including a font resource.
`fTgtFileName`	ScriptCheck passes you the file name contained in the target file spec.
`fTgtFinderAttrs`	Return the Finder flags of the original file.
`fTgtDataForkSize`	Return the size in bytes of the original data fork of the file
`fTgtRsrcForkSize`	Return the size in bytes of the original resource fork of the file.
`fTgtCreationDate`	Return the creation date of the original file.

`fTgtFileType`	Return the file type of the original file.
`fTgtFileCreator`	Return the file creator of the original file.
`fTgtModDate`	Return the modification date of the original file.
`fSrcRsrcType`	ScriptCheck passes you the resource type of the source resource.
`fSrcRsrcID`	ScriptCheck passes you the resource ID of the source resource.
`fTgtRsrcType`	ScriptCheck passes you the resource type of the target resource.
`fTgtRsrcID`	ScriptCheck passes you the resource ID of the target resource.
`fTgtRsrcAttrs`	Return the resource attributes of the original resource.
`fTgtRsrcSize`	Return the size in bytes of the original resource.

Version Compare Extension Parameter Block

The version info function code resource is passed a pointer to a parameter block containing fields that you will pass back original source information.

```
#define  kVersTypeIsFile   -1
#define  kVersTypeIsRsrc   1

typedef struct {
 ->   FSSpec      fSrcFSSpec;
 ->   long        fSrcDataType;
 ->   Str31       fTgtFileName;
<-    long        fTgtVersionNum;
} TgtFileVersType;

typedef struct {
 ->   FSSpec      fSrcFSSpec;
 ->   long        fSrcDataType;
 ->   OSType      fSrcRsrcType;
 ->   short       fSrcRsrcID;
 ->   OSType      fTgtRsrcType;
 ->   short       fTgtRsrcID;
<-    long        fTgtRsrcVersionNum;
} TgtRsrcVersType;

typedef union {
   TgtFileVersType         fFileVers;
   TgtRsrcVersType         fRsrcVers;
} TargetVersPB, *TargetVersPBPtr;
```

Field descriptions

`fSrcFSSpec`	A FSSPec to the source file referenced by the atom that ScriptCheck supplies you.
`fSrcDataType`	ScriptCheck supplies one of two values: `kDataTypeIsFile` if ScriptCheck wants information about a file; otherwise, `kDataTypeIsRsrc` if ScriptCheck wants information about a resource, including a font resource.
`fTgtFileName`	ScriptCheck passes you the file name contained in the target file spec.

`fTgtVersionNum`	Return the version number of the original file.
`fSrcRsrcType`	ScriptCheck passes you the resource type of the source resource.
`fSrcRsrcID`	ScriptCheck passes you the resource ID of the source resource.
`fTgtRsrcType`	ScriptCheck passes you the resource type of the target resource.
`fTgtRsrcID`	ScriptCheck passes you the resource ID of the target resource.
`fTgtRsrcVersionNum`	Return the version number of the original resource.

Resource Description

The templates of the ScriptCheck Extension script resources are shown below. The following definition is contained in the file “ScriptCheckTypes.r”.

File/Resource Info Extension Resource

```
#define  scriptCheckAtomExtFlags                    \
      fill bit[16];

type 'scex' {
      switch {
         case format0:
            key integer = 0;
            scriptCheckAtomExtFlags;   /* Flags for Format 0 */
            literal longint;           /* Type of code resource */
            integer;                   /* ID of code resource */
            longint;                   /* Refcon for parameter block */
            longint;                   /* Requested Memory in bytes */
            evenPaddedString;          /* Summary Description  */
      };
};
```

Field descriptions

`Flags`	Currently reserved for use by Apple Computer, Inc. (2-bytes)
`Code Rsrc Type`	The resource type of the file / resource info code resource. (4-bytes)
`Code Rsrc ID`	The resource ID of the file / resource info code resource. (2-bytes)
`RefCon`	Currently Ignored. (4-bytes)
`Requested Memory`	The minimum number of free bytes the file / resource info code resource needs during execution. Currently Ignored. (4-bytes)
`Summary Description`	An optional string briefly describing the purpose of this file / resource info function. This string is never displayed to the user. (even-padded Pascal string)

Version Info Extension Resource

```
#define  scriptCheckVersCompareFlags                \
      fill bit[16];
```

```
type 'scvc' {
      switch {
         case format0:
            key integer = 0;              /* Format version */
            scriptCheckVersCompareFlags;  /* Flags for Format 0 */
            literal longint;              /* Type of code resource */
            integer;                      /* ID of code resource */
            longint;                      /* Refcon for param block */
            longint;                      /* Requested Memory in
bytes*/
            evenPaddedString;             /* Status Description  */

      };
};
```

Field descriptions

`Flags`	Currently reserved for use by Apple Computer, Inc. (2-bytes)
`Code Rsrc Type`	The resource type of the version info code resource. (4-bytes)
`Code Rsrc ID`	The resource ID of the version info code resource. (2-bytes)
`RefCon`	Currently Ignored. (4-bytes)
`Requested Memory`	The minimum number of free bytes the version info code resource needs during execution. Currently Ignored. (4-bytes)
`Summary Description`	An optional string briefly describing the purpose of this version info function. This string is never displayed to the user. (even-padded Pascal string)