

[0064] The result of this intermediate step of the package-creation process is a number of files, shown in FIG. 2, for each package that is to be constructed, including:

Package definition file (pkd) 224;
 Component mapping file (cpm) 206;
 Component relations file (crf) 226 OPTIONAL;
 Build manifest file 202 OPTIONAL;
 Registry file (rgu) 210 OPTIONAL; and
 XML settings file 222; OPTIONAL

[0065] From these files, (including validation, e.g., via XSD 203 and XSD 225) a package generation process 230 constructs a final package file 232, generally in accordance with the flowchart shown in FIG. 6. As generally represented in FIG. 6, after some checks and validations (steps 600-610), a package collection is created from the packages (step 612), including mapping each package to the package definition, reading the build manifest file 208 for that package and generating the package from that data (steps 618-622).

[0066] The process of converting the build manifest file into a final package file list form and processing each executable with a relmerge tool 250 (the tool that inserts relocation information into an executable as described below with reference to FIG. 8) is generally depicted in FIGS. 7A and 7B. As generally represented in FIGS. 7A and 7B via steps 700-714, a directive file is created and the build manifest file is located and parsed, (assuming no errors occurred). If the parsing is successful (step 716), a device manifest object is created at step 720, and the process continues to step 726 of FIG. 7B, which ensures that the device manifest object was properly created.

[0067] Via steps 732 and 756, each file listed in the build manifest file is processed. To this end, at step 734 the file is found and determined whether to be executable at step 738. If not, the file is copied as is to a temporary build directory, otherwise the file needs to be processed by a tool 250 (FIG. 2, e.g., named relmerge.exe) that enables executable code relocation/fix-up on the device at install time (if not already processed, as tested via step 740). If the file needs relmerge processing, described below with reference to FIGS. 8A and 8B, the tool is called at step 744, and if it executes successfully, the filename is added to the device manifest.

[0068] Thus, as described above with reference to FIG. 7A and 7B, the file contents for a package, listed in the build manifest file 208, are reviewed and any executable code is processed prior to insertion into the package to enable executable code relocation/fix-up on the device at install time. To convert the build manifest file into a final package file list form, each executable is processed with the relmerge tool 250 that compresses the relocation information that is already in the file, and optionally (if a .REL file is provided) provides more detailed relocation information that allows the operating system to separate the code and data sections into discontinuous memory regions.

[0069] The relmerge tool operations are generally depicted in the flow diagram of FIG. 8. As represented by step 802, a copy of the input file is made, because the input file will be modified by running this too. The relmerge tool 250 makes a thus copy of the input file and works from that copy, which is deleted when the program exits.

[0070] At step 804, the signature is removed, because the signature would otherwise cause a later portion of the process (a space accounting check, described below) to fail. Note that since the tool 250 will be outputting a completely different file, the signature would have no relevance to the output file in any event. Note that in a portable executable file, (the file format for .EXE and .DLL files), the signature is stored at the end of the file outside of any section in the file, wherein each section corresponds to a unit with which data is loaded into memory by the loader. Each section has a header, called an 032 header, or identified with the IMAGE_SECTION_HEADER structure. As generally represented by step 806, to facilitate manipulating the file, the tool 250 parses the PE file headers for the PE file and its sections into suitable internal data structures. The IMAGE_SECTION_HEADER structure represents the image section header format (additional details about the IMAGE_SECTION_HEADER may be found at msdn.microsoft.com):

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER,
*PIMAGE_SECTION_HEADER;
```

[0071] Because the file will be re-layed out with new headers and with padding removed, a space accounting check (step 808) is performed to verify that there is no information in the file that is not included in the sections of the file. Note that there are some situations where data is stored in an .EXE or .DLL file outside of the Sections, including file signatures and Codeview debug directory entries, (which are handled by the tool 250). There are other applications that may store data outside of a section, such as a self-extracting executable file which stores the data to be extracted after the .EXE itself. The compressed data is not stored in a section, otherwise the loader would attempt to load the compressed data into memory, which is not how a self-extracting executable should operate. The tool does not support these instances.

[0072] Space Accounting is implemented by an instance of the CSpaceAccounting class, which maintains an array of SpaceBlock structures, each of which accounts for block of data in the file. To implement Space Accounting, the areas in the file that can be accounted for are added to the SpaceAccounting as individual blocks. This is done for the file headers (including the E32 and O32 headers), and for each of the sections in the file. To accommodate CodeView debug entries, each of those are also added as a separate block. The blocks are then sorted by their offset within the file. Blocks which are adjacent in the new ordering (e.g., Block 2 begins on the byte immediately after the last byte of Block 1) are merged. At the end of the process, if all of the