

space in the file can be accounted for, then the list should contain one and only one block, which starts at offset 0 and has a length which is the length of the entire file. If that condition is true, then the test passes. If not, an error is reported and the tool exits.

[0073] If the space accounting check passes, the tool 250 searches for the .REL file in the same directory where the input file is specified, as represented by step 810. If so, it is processed via step 812, otherwise the relocation section of the PE file is processed via step 814. More particularly, the relocation parsing tool can parse two different types of relocation information, namely the relocation information in a .REL file, which contains destination section information, and the relocation information in the .reloc section of a PE file, which does not contain destination section information. To this end, relocation parsing has two entry points corresponding to parsing either a .REL file or the internal relocations in a PE file; in a current implementation, it is legal to call either one or the other, but not both.

[0074] In order to store the relocations, a two dimensional array of CRelocData classes is maintained, wherein the two dimensions of this array are the source section and the destination section for the relocation in question. The source section for a relocation is the section where the relocation is located, wherein a relocation is an instruction to update a particular piece of data in the file when the file is fixed up to load at a particular address; the source section identifies in which section that piece of data occurs. The source section is inferred by looking at the relative virtual address of the relocation and comparing that with the relative virtual address ranges of the sections in the file.

[0075] The destination section for a relocation identifies which section contains the piece of data to which the relocation is pointing. An inspection technique does not always work, because optimizers have been known to optimize the code in such a way that the relocations appear to point to other sections, e.g., it will optimize out an addition or subtraction and place it in the reference instead of in the code. For this reason, the destination section is not inferred from the data. The difference between the two relocation formats (.REL or .reloc) is whether the destination section for the relocation is known. The .REL file explicitly identifies the destination section, while the .reloc section does not.

[0076] As a result, for files which only have the relocation information in the .reloc section, the entire file needs to be relocated together. Without destination section information, it is not possible to separate two sections and relocate them by different amounts. Thus, this tool needs to keep track of both the source and destination section for each relocation, and keep track of whether the destination sections are valid.

[0077] To do this, the process maintains a two dimensional array of CRelocData classes, and each CRelocData class builds its own stream of relocation data. This array is fixed size in both dimensions, meaning that the tool can only handle PE files with some maximum (e.g., sixteen) sections. With that limitation, the data format for persisting the relocations stores eight bits for each of the source and destination sections, leaving the possibility in the data format for 256 sections. Two other functions (CalculateRelocSize and WriteRelocationsToFile) then combine these

individual streams by writing out a block header for each combination of source and destination section which has at least one relocation.

[0078] Relocation encoding is implemented in the CRelocData class. This class takes a stream of relocation addresses (as individual calls to the CRelocData::AddReloc method), and creates a byte stream which represents the commands necessary to encode those relocations. That byte stream can be retrieved later. To implement this, the class effectively stays one command behind, always starting out with a "Single" command, representing a single relocation. Then as new relocations arrive (at the AddReloc method), they are analyzed to see if a pattern can be formed using the previous command and the new location. If the previous command is a "Single" command, the previous command can only be extended by transforming it from a "Single" command to a "Pattern" command if the new relocation address is DWORD aligned, and is within the maximum skip range of the pattern command (which is 3 DWORDs). If the previous command is a pattern command, then that pattern already has an established form, and the next element in the pattern can be deduced. If the new address happens to match the next element in that pattern, the pattern is extended by one cycle. Otherwise, a new Single command is started.

[0079] Returning to the overall flow of FIG. 8, because the sections of the file occur linearly in the file following the headers, the entire file needs to be re-layed out to accommodate the new header format and the removal of all of the padding between the sections. To this end, as represented by step 818, the old .RELOC section (if one existed) is removed, and the new .CRELOC section is added at the end, if there are any relocations to add. Using the file layout completed in step 818, the output file is finally created at step 820, writing the headers then the data for each section to the output file.

[0080] At step 822, relmerge.exe now tests to see if the target output file is nk.exe. If the output file is nk.exe, then the relmerge tool 250 processes the contents of two files, to write out the pTOC information and RomExt information, (described below). The relmerge tool 250 looks at the output file because nk.exe is created by copying different files based on debug settings. The first file processed is config.bsm.xml. This file is produced by MakePkg.exe during Image Update processing. It contains a textual representation of the names and desired values for the FIXUPVARS in the system kernel. The contents of this file are parsed and stored for later use. The second file is the map file for the input file. This file is processed by ProcessFixupVars. It takes the source file path, copies it and replaces the .exe with .map and attempts to open the map file (map files are text files containing a large amount of information about the physical and virtual addresses of functions and variables within the PE file). If the map file is successfully opened, the first line is parsed to retrieve the timestamp of the map file. The timestamp is then compared against the timestamp in the PE file. If they are different, a warning is produced and map file processing stops. If the timestamps match, each line of the file is read and a regular expression string is used to look for each FIXUPVAR. If a match is found, the address information is taken from the map file and is used to write the new variable value (from config.bsm.xml) into the source file at the correct location. At the same time, ProcessFixupVars