(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
13 February 2003 (13.02.2003)

PCT

(10) International Publication Number
WO 03/012639 A2

(51) International Patent Classification[7]: G06F 9/445

(21) International Application Number: PCT/IB02/02889

(22) International Filing Date: 23 July 2002 (23.07.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

09/917,026	26 July 2001 (26.07.2001)	US
09/916,460	26 July 2001 (26.07.2001)	US
09/916,900	26 July 2001 (26.07.2001)	US
09/927,131	10 August 2001 (10.08.2001)	US
09/969,305	2 October 2001 (02.10.2001)	US

(71) Applicant: **KYOCERA WIRELESS CORPORATION** [US/US]; 10300 Campus Point Drive, San Diego, CA 92121 (US).

(72) Inventors: **RAJARAM, Gowri**; 3520 LEbon Drive, Apt. 5330, San Diego, CA 92122 (US). **SECKENDORF, Paul**; 14074 Davenport Avenue, San Diego, CA 92129 (US). **KAPLAN, Diego**; 5288 Soledad Mt. Road, San Diego, CA 92109 (US).

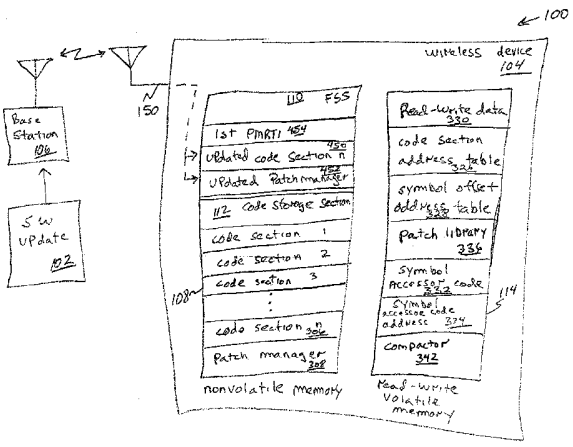
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR COMPACTING FIELD UPGRADEABLE WIRELESS COMMUNICATION DEVICE SOFTWARE CODE SECTIONS



(57) Abstract: A system and method are provided for reorganizing software instructions stored in a wireless communications device memory. The method comprises: storing wireless device system software in a plurality of current code sections with the start of code sections at corresponding start addresses by creating a second plurality of contiguously addressed memory blocks, identifying each memory block with a corresponding code section, and storing code sections in identified memory blocks; receiving a new code section via a wireless communications device air interface; identifying a current code section for updating; calculating the code section sizes; in response to calculating the code section sizes, generating a compaction schedule; resizing current code sections; following the resizing of the current code sections, changing the code section start addresses; temporarily moving code sections into a file system section; replacing the identified current code section with the new code section by storing the code sections from the file system section into memory blocks to maintain contiguous addressing, in response to the compaction schedule; and, executing the updated system software.

WO 03/012639 A2

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

SYSTEM AND METHOD FOR COMPACTING FIELD UPGRADEABLE WIRELESS COMMUNICATION DEVICE SOFTWARE CODE SECTIONS

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention generally relates to wireless communications devices and, more particularly, to a system and method for compacting code sections in the system software of a wireless communication device in the field, updated via an airlink interface.

2. Description of the Related Art

It is not uncommon to release software updates for phones that are already in the field. These updates may relate to problems found in the software once the phones have been manufactured and distributed to the public. Some updates may involve the use of new features on the phone, or services provided by the service provider. Yet other updates may involve regional problems, or problems associated with certain carriers. For example, in certain regions the network layout of carriers may impose airlink interface conditions on the handset that cause the handset to demonstrate unexpected behavior such as improper channel searching, improper call termination, improper audio, or the like.

The traditional approach to such updates has been to recall the wireless communications device, also referred to herein as a wireless device, phone, telephone, or handset, to the nearest carrier retail/service outlet, or to the manufacturer to process the changes. The costs involved in such updates are extensive and eat into the bottom line. Further, the customer is inconvenienced and likely to be irritated. Often times, the practical solution is to issue the customer new phones.

It would be advantageous if wireless communications device software could be upgraded cheaply, and without inconvenience to the customer.

It would be advantageous if wireless communications device software could be upgraded without the customer losing the use of their phones for a significant period of time.

It would be advantageous if wireless communications device software could be updated with a minimum of technician service time, or without the need to send the device into a service facility.

It would be advantageous if the wireless device system software could be differentiated into code sections, so that only specific code sections of system software would need to be replaced, to update the system software. It would be advantageous if these code sections could be communicated to the wireless device via the airlink.

It would be advantageous if the system software could be updated with code sections larger in size than the currently residing code sections. It would also be advantageous if the system software could be rearranged to accommodate these larger updated code sections.

SUMMARY OF THE INVENTION

Wireless communications device software updates give customers the best possible product and user experience. An expensive component of the business involves the recall of handsets to update the software. These updates may be necessary to offer the user additional services or to address problems discovered in the use of the phone after it has been manufactured. The present invention makes it possible to practically upgrade handset software in the field, via the airlink interface, when the upgrade code sections are larger than the code sections they are replacing.

Accordingly, a method is provided for reorganizing software instructions stored in a wireless communications device memory. The method comprises: storing wireless device system software in a plurality of current code sections with the start of code sections at corresponding start addresses by creating a second plurality of contiguously addressed memory blocks, identifying each memory block with a corresponding code section, and storing code sections in identified memory blocks; receiving a new code section via a wireless communications device air interface; identifying a current code section for updating; calculating the code section sizes; in response to calculating the code section sizes, generating a compaction schedule; resizing current code sections; following the resizing of the current code sections, changing the code section start addresses; temporarily moving code sections into a file system section; replacing the identified current code section with the new code section by storing the code sections from the file system section into memory blocks to maintain contiguous addressing, in response to the compaction schedule; and, executing the updated system software.

Additional details of the above-described method for reorganizing software instructions in wireless device system software, and a wireless device system for reorganizing software instructions are presented in detail below.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a schematic block diagram of the overall wireless device software maintenancesystem.

Fig. 2 is a schematic block diagram of the software maintenance system, highlighting the installation of instruction sets via the airlink interface.

Fig. 3 is a schematic block diagram illustrating the present invention system for reorganizing software instructions in a wireless communications device.

Fig. 4 is a schematic block diagram of the wireless device memory.

Fig. 5 is a table representing the code section address table of Fig. 3.

Fig. 6 is a detailed depiction of symbol library one of Fig. 3, with symbols.

Fig. 7 is a table representing the symbol offset address table of Fig. 3.

Fig. 8 is a schematic block diagram illustrating the results of exemplary compacting operation.

Figs. 9a through 9e illustrate the operation of an exemplary compaction schedule.

Figs. 10a through 10b are a flowchart illustrating the present invention method for reorganizing software instructions stored in a wireless communications device memory.

Fig. 11 is a flowchart illustrating an alternate aspect of the method of Figs. 10a through 10b.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Some portions of the detailed descriptions that follow are presented in terms of procedures, steps, logic blocks, codes, processing, and other symbolic representations of operations on data bits within a wireless device microprocessor or memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. A procedure, microprocessor executed step, application, logic block, process, etc., is here, and generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated in a microprocessor based wireless device. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. Where physical devices, such as a memory are mentioned, they are connected to other physical devices through a bus or other electrical connection. These physical devices can be considered to interact with logical processes or applications and, therefore, are "connected" to logical operations. For example, a memory can store or access code to further a logical operation, or an application can call a code section from memory for execution.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "connecting" or "translating" or "displaying" or "prompting" or "determining" or "displaying" or "recognizing" or the like, refer to the action and processes of in a wireless device microprocessor system that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the wireless device memories or registers or other such information storage, transmission or display devices.

Fig. 1 is a schematic block diagram of the overall wireless device software maintenance system 100. The present invention system software organization is presented in detail below, following a general overview of the software maintenance system 100. The general system 100 describes a process of delivering system software updates and instruction sets (programs), and installing the delivered software in a wireless device. System software updates or patch manager run time instructions (PMRTI) that are

more generally known as instruction sets, are created by the manufacturer of the handsets. The system software is organized into symbol libraries. The symbol libraries are arranged into code sections. When symbol libraries are to be updated, the software update 102 is transported as one or more code sections. The software update is broadcast to wireless devices in the field, of which wireless communications device 104 is representative, or transmitted in separate communications from a base station 106 using well known, conventional air, data or message transport protocols. The invention is not limited to any particular transportation format, as the wireless communications device can be easily modified to process any available over-the-air transport protocol for the purpose of receiving system software and PMRTI updates.

The system software can be viewed as a collection of different subsystems. Code objects can be tightly coupled into one of these abstract subsystems and the resulting collection can be labeled as a symbol library. This provides a logical breakdown of the code base and software patches and fixes can be associated with one of these symbol libraries. In most cases, a single update is associated with one, or at most two, symbol libraries. The rest of the code base, the other symbol libraries, remain unchanged. The notion of symbol libraries provides a mechanism to deal with code and constants. The read-write (RW) data, on the other hand, fits into a unique individual RW library that contains RAM based data for all libraries.

Once received by the wireless device 104, the transported code section must be processed. This wireless device over-writes a specific code section of nonvolatile memory 108. The nonvolatile memory 108 includes a file system section (FSS) 110 and a code storage section 112. The code section is typically compressed before transport in order to minimize occupancy in the FSS 110. Often the updated code section will be accompanied by its RW data, which is another kind of symbol library that contains all the RW data for each symbol library. Although loaded in random access volatile read-write memory 114 when the system software is executing, the RW data always needs to be stored in the nonvolatile memory 108, so it can be loaded into random access volatile read-write memory 114 each time the wireless device is reset. This includes the first time RW data is loaded into random access volatile read-write memory. As explained in more detail below, the RW data is typically arranged with a patch manager code section.

The system 100 includes the concept of virtual tables. Using such tables, symbol libraries in one code section can be patched (replaced), without breaking (replacing) other parts of the system software (other code sections). Virtual tables execute from random access volatile read-write memory 114 for efficiency purposes. A code section address table and symbol offset address table are virtual tables.

The updated code sections are received by the wireless device 104 and stored in the FSS 110. A wireless device user interface (UI) will typically notify the user that new software is available. In response to UI prompts the user acknowledges the notification and signals the patching or updating operation. Alternately, the updating operation is performed automatically. The wireless device may be

unable to perform standard communication tasks as the updating process is performed. The patch manager code section includes a non-volatile read-write driver symbol library that is also loaded into random access volatile read-write memory 114. The non-volatile read-write driver symbol library causes code sections to be overwritten with updated code sections. The patch manager code section includes the read-write data, code section address table, and symbol offset address table, as well a symbol accessor code and the symbol accessor code address (discussed below). Portions of this data are invalid when updated code sections are introduced, and an updated patch manager code sections includes read-write data, a code section address table, and a symbol offset address table valid for the updated code sections. Once the updated code sections are loaded into the code storage section 112, the wireless device is reset. Following the reset operation, the wireless device can execute the updated system software. It should also be understood that the patch manager code section may include other symbol libraries that have not been discussed above. These other symbol libraries need not be loaded into read-write volatile memory 114.

Fig. 2 is a schematic block diagram of the software maintenance system 100, highlighting the installation of instruction sets via the airlink interface. In addition to updating system software code sections, the maintenance system 100 can download and install instructions sets, programs, or patch manager instruction sets (PMIS), referred to herein as patch manager run time instructions (PMRTI). The PMRTI code section 200 is transported to the wireless device 104 in the same manner as the above-described system software code sections. PMRTI code sections are initially stored in the FSS 110. A PMRTI code section is typically a binary file that may be visualized as compiled instructions to the handset. A PMRTI code section is comprehensive enough to provide for the performance of basic mathematical operations and the performance of conditionally executed operations. For example, an RF calibration PMRTI could perform the following operations:

IF RF CAL ITEM IS LESS THAN X
EXECUTE INSTRUCTION
ELSE
EXECUTE INSTRUCTION

A PMRTI can support basic mathematical operations, such as: addition, subtraction, multiplication, and division. As with the system software code sections, the PMRTI code section may be loaded in response to UI prompts, and the wireless device must be reset after the PMRTI is loaded into code storage section 112. Then the PMRTI section can be executed. If the PMRTI code section is associated with any virtual tables or read-write data, an updated patch manager code section will be transported with the PMRTI for installation in the code storage section 112. Alternately, the PMRTI can be kept and processed from the FSS 110. After the handset 104 has executed all the instructions in the PMRTI section, the PMRTI section can be deleted from the FSS 110.

PMRTI is a very powerful runtime instruction engine. The handset can execute any instruction delivered to it through the PMRTI environment. This mechanism may be used to support RF calibrations and PRI updates. More generally, PMRTI can be used to remote debug wireless device software when software problems are recognized by the manufacturer or service provider, typically as the result of user complaints. PMRTI can also record data needed to diagnose software problems. PMRTI can launch newly downloaded system applications for data analysis, debugging, and fixes. PMRTI can provide RW data based updates for analysis and possible short term fix to a problem in lieu of an updated system software code section. PMRTI can provide memory compaction algorithms for use by the wireless device.

In some aspects of the invention, the organization of the system software into symbol libraries may impact the size of the volatile memory 114 and nonvolatile memory 108 required for execution. This is due to the fact that the code sections are typically larger than the symbol libraries arranged in the code sections. These larger code sections exist to accommodate updated code sections. Organizing the system software as a collection of libraries impacts the nonvolatile memory size requirement. For the same code size, the amount of nonvolatile memory used will be higher due to the fact that code sections can be sized to be larger than the symbol libraries arranged within.

Once software updates have been delivered to the wireless device, the software maintenance system 100 supports memory compaction. Memory compaction is similar to disk de-fragmentation applications in desktop computers. The compaction mechanism ensures that memory is optimally used and is well balanced for future code section updates, where the size of the updated code sections are unpredictable. The system 100 analyzes the code storage section as it is being patched (updated). The system 100 attempts to fit updated code sections into the memory space occupied by the code section being replaced. If the updated code section is larger than the code section being replaced, the system 100 compacts the code sections in memory 112. Alternately, the compaction can be calculated by the manufacturer or service provider, and compaction instructions can be transported to the wireless device 104.

Compaction can be a time consuming process owing to the complexity of the algorithm and also the vast volume of data movement. The compaction algorithm predicts feasibility before it begins any processing. UI prompts can be used to apply for permission from the user before the compaction is attempted.

In some aspects of the invention, all the system software code sections can be updated simultaneously. A complete system software upgrade, however, would require a larger FSS 110.
Fig. 3 is a schematic block diagram illustrating the present invention system for reorganizing software instructions in a wireless communications device. The system 300 comprises a code storage section 112 in memory 108 including executable wireless device system software differentiated into a plurality of

current code sections. Code section one (302), code section two (304), code section n (306), and a patch manager code section 308 are shown. However, the invention is not limited to any particular number of code sections. Further, the system 300 further comprises a first plurality of symbol libraries arranged into the second plurality of code sections. Shown are symbol library one (310) arranged in code section one (302), symbol libraries two (312) and three (314) arranged in code section two (304), and symbol library m (316) arranged in code section n (306). Each library comprises symbols having related functionality. For example, symbol library one (310) may be involved in the operation of the wireless device liquid crystal display (LCD). Then, the symbols would be associated with display functions. As explained in detail below, additional symbol libraries are arranged in the patch manger code section 308.

Fig. 4 is a schematic block diagram of the wireless device memory. As shown, the memory is the code storage section 112 of Fig. 1. The memory is a writeable, nonvolatile memory, such as Flash memory. It should be understood that the code sections need not necessarily be stored in the same memory as the FSS 110. It should also be understood that the present invention system software structure could be enabled with code sections stored in a plurality of cooperating memories. The code storage section 112 includes a second plurality of contiguously addressed memory blocks, where each memory block stores a corresponding code section from the second plurality of code sections. Thus, code section one (302) is stored in a first memory block 400, code section two (304) in the second memory block 402, code section n (306) in the nth memory block 404, and the patch manager code section (308) in the pth memory block 406.

Contrasting Figs. 3 and 4, the start of each code section is stored at corresponding start addresses in memory, and symbol libraries are arranged to start at the start of code sections. That is, each symbol library begins at a first address and runs through a range of addresses in sequence from the first address. For example, code section one (302) starts at the first start address 408 (marked with "S") in code storage section memory 112. In Fig. 3, symbol library one (310) starts at the start 318 of the first code section. Likewise code section two (304) starts at a second start address 410 (Fig. 4), and symbol library two starts at the start 320 of code section two (Fig. 3). Code section n (306) starts at a third start address 412 in code storage section memory 112 (Fig. 4), and symbol library m (316) starts at the start of code section n 322 (Fig. 3). The patch manager code section starts at pth start address 414 in code storage section memory 112, and the first symbol library in the patch manager code section 308 starts at the start 324 of the patch manager code section. Thus, symbol library one (310) is ultimately stored in the first memory block 400. If a code section includes a plurality of symbol libraries, such as code section two (304), the plurality of symbol libraries are stored in the corresponding memory block, in this case the second memory block 402.

In Fig. 3, the system software structure 300 further comprises a code section address table 326 as a type of symbol included in a symbol library arranged in the patch manager code section 308. The code

section address table cross-references code section identifiers with corresponding code section start addresses in memory.

Fig. 5 is a table representing the code section address table 326 of Fig. 3. The code section address table 326 is consulted to find the code section start address for a symbol library. For example, the system 300 seeks code section one when a symbol in symbol library one is required for execution. To find the start address of code section one, and therefore locate the symbol in symbol library one, the code section address table 326 is consulted. The arrangement of symbol libraries in code sections, and the tracking of code sections with a table permits the code sections to be moved or expanded. The expansion or movement operations may be needed to install upgraded code sections (with upgraded symbol libraries).

Returning to Fig. 3, it should be noted that not every symbol library necessarily starts at the start of a code section. As shown, symbol library three (314) is arranged in code section two (304), but does not start of the code section start address 320. Thus, if a symbol in symbol library three (314) is required for execution, the system 300 consults the code section address table 326 for the start address of code section two (304). As explained below, a symbol offset address table permits the symbols in symbol library three (314) to be located. It does not matter that the symbols are spread across multiple libraries, as long as they are retained with the same code section.

As noted above, each symbol library includes functionally related symbols. A symbol is a programmer-defined name for locating and using a routine body, variable, or data structure. Thus, a symbol can be an address or a value. Symbols can be internal or external. Internal symbols are not visible beyond the scope of the current code section. More specifically, they are not sought by other symbol libraries, in other code sections. External symbols are used and invoked across code sections and are sought by libraries in different code sections. The symbol offset address table typically includes a list of all external symbols.

For example, symbol library one (310) may generate characters on a wireless device display. Symbols in this library would, in turn, generate telephone numbers, names, the time, or other display features. Each feature is generated with routines, referred to herein as a symbol. For example, one symbol in symbol library one (310) generates telephone numbers on the display. This symbol is represented by an "X", and is external. When the wireless device receives a phone call and the caller ID service is activated, the system must execute the "X" symbol to generate the number on the display. Therefore, the system must locate the "X" symbol.

Fig. 6 is a detailed depiction of symbol library one (310) of Fig. 3, with symbols. Symbols are arranged to be offset from respective code section start addresses. In many circumstances, the start of the symbol library is the start of a code section, but this is not true if a code section includes more than one symbol library. Symbol library one (310) starts at the start of code section one (see Fig. 3). As shown in

Fig. 6, the "X" symbol is located at an offset of (03) from the start of the symbol library and the "Y" symbol is located at an offset of (15). The symbol offset addresses are stored in a symbol offset address table 328 in the patch manager code section (see Fig. 3).

Fig. 7 is a table representing the symbol offset address table 328 of Fig. 3. The symbol offset address table 328 cross-references symbol identifiers with corresponding offset addresses, and with corresponding code section identifiers in memory. Thus, when the system seeks to execute the "X" symbol in symbol library one, the symbol offset address table 328 is consulted to located the exact address of the symbol, with respect to the code section in which it is arranged.

Returning to Fig. 3, the first plurality of symbol libraries typically all include read-write data that must be consulted or set in the execution of these symbol libraries. For example, a symbol library may include an operation dependent upon a conditional statement. The read-write data section is consulted to determine the status required to complete the conditional statement. The present invention groups the read-write data from all the symbol libraries into a shared read-write section. In some aspects of the invention, the read-write data 330 is arranged in the patch manager code section 308. Alternately (not shown), the read-write data can be arranged in a different code section, code section n (306), for example.

The first plurality of symbol libraries also includes symbol accessor code arranged in a code section to calculate the address of a sought symbol. The symbol accessor code can be arranged and stored at an address in a separate code section, code section two (304), for example. However, as shown, the symbol accessor code 332 is arranged and stored at an address in the patch manager code section 308. The system software structure 300 further comprises a first location for storage of the symbol accessor code address. The first location can be a code section in the code storage section 112, or in a separate memory section of the wireless device (not shown). The first location can also be arranged in the same code section as the read-write data. As shown, the first location 334 is stored in the patch manager code section 308 with the read-write data 330, the symbol offset address table 328, the code section address table 326, and the symbol accessor code 332, and the patch library (patch symbol library) 336.

The symbol accessor code accesses the code section address table and symbol offset address tables to calculate, or find the address of a sought symbol in memory. That is, the symbol accessor code calculates the address of the sought symbol using a corresponding symbol identifier and a corresponding code section identifier. For example, if the "X" symbol in symbol library one is sought, the symbol accessor is invoked to seek the symbol identifier (symbol ID) X_1, corresponding to the "X" symbol (see Fig. 7). The symbol accessor code consults the symbol offset address table to determine that the X_1 symbol identifier has an offset of (03) from the start of code section one (see Fig. 6). The symbol accessor code is invoked to seek the code section identifier CS_1, corresponding to code section one. The symbol accessor code consults the code section address table to determine the start address associated with code

section identifier (code section ID) CS_1. In this manner, the symbol accessor code determines that the symbol identifier X_1 is offset (03) from the address of (00100), or is located at address (00103).

The symbol "X" is a reserved name since it is a part of the actual code. In other words, it has an absolute data associated with it. The data may be an address or a value. The symbol identifier is an alias created to track the symbol. The symbol offset address table and the code section address table both work with identifiers to avoid confusion with reserved symbol and code section names. It is also possible that the same symbol name is used across many symbol libraries. The use of identifiers prevents confusion between these symbols.

Returning to Fig. 1, the system software structure 300 further comprises a read-write volatile memory 114, typically random access memory (RAM). The read-write data 330, code section address table 326, the symbol offset address table 328, the symbol accessor code 332, and the symbol accessor code address 334 are loaded into the read-write volatile memory 114 from the patch manager code section for access during execution of the system software. As is well known, the access times for code stored in RAM is significantly less than the access to a nonvolatile memory such as Flash.

Returning to Fig. 3, it can be noted that the symbol libraries need not necessarily fill the code sections into which they are arranged, although the memory blocks are sized to exactly accommodate the corresponding code sections stored within. Alternately stated, each of the second plurality of code sections has a size in bytes that accommodates the arranged symbol libraries, and each of the contiguously addressed memory blocks have a size in bytes that accommodates corresponding code sections. For example, code section one (302) may be a 100 byte section to accommodate a symbol library having a length of 100 bytes. The first memory block would be 100 bytes to match the byte size of code section one. However, the symbol library loaded into code section 1 may be smaller than 100 bytes. As shown in Fig. 3, code section one (302) has an unused section 340, as symbol library one (310) is less than 100 bytes. Thus, each of the second plurality of code sections may have a size larger than the size needed to accommodate the arranged symbol libraries. By "oversizing" the code sections, larger updated symbol libraries can be accommodated.

Contiguously addressed memory blocks refers to partitioning the physical memory space into logical blocks of variable size. Code sections and memory blocks are terms that are essentially interchangeable when the code section is stored in memory. The concept of a code section is used to identify a section of code that is perhaps larger than the symbol library, or the collection of symbol libraries in the code section as it is moved and manipulated.

As seen in Fig. 3, the system 300 includes a patch symbol library, which will be referred to herein as patch library 336, to arrange new code sections in the code storage section with the current code sections. The arrangement of new code sections with current code sections in the code storage section

forms updated executable system software. The patch manager 336 not only arranges new code sections in with the current code sections, it also replaces code sections with updated code sections.

Returning to Fig. 4, the file system section 110 of memory 108 receives new code sections, such as new code section 450 and updated patch manager code section 452. The file system section also receives a first patch manager run time instruction (PMRTI) 454 including instructions for arranging the new code sections with the current code sections. As seen in Fig. 1, an airlink interface 150 receives new, or updated code sections, as well as the first PMRTI. Although the airlink interface 150 is being represented by an antenna, it should be understood that the airlink interface would also include an RF transceiver, baseband circuitry, and demodulation circuitry (not shown). The file system section 110 stores the new code sections received via the airlink interface 150. The patch library 336, executing from read-write volatile memory 114, replaces a first code section in the code storage section, code section n (306) for example, with the new, or updated code section 450, in response to the first PMRTI 454. Typically, the patch manager code section 308 is replaced with the updated patch manager code section 452. When code sections are being replaced, the patch library 336 over-writes the first code section, code section n (306) for example, in the code storage section 112 with the updated code sections, code section 450 for example, in the file system section 110. In the extreme case, all the code sections in code storage section 112 are replaced with updated code sections. That is, the FSS 110 receives a second plurality of updated code sections (not shown), and the patch library 336 replaces the second plurality of code sections in the code storage section 112 with the second plurality of updated code sections. Of course, the FSS 110 must be large enough to accommodate the second plurality of updated code sections received via the airlink interface.

As noted above, the updated code sections being received may include read-write data code sections, code section address table code sections, symbol libraries, symbol offset address table code sections, symbol accessor code sections, or a code section with a new patch library. All these code sections, with their associated symbol libraries and symbols, may be stored as distinct and independent code sections. Then each of these code sections would be replaced with a unique updated code section. That is, an updated read-write code section would be received and would replace the read-write code section in the code storage section. An updated code section address table code section would be received and would replace the code section address table code section in the code storage section. An updated symbol offset address table code section would be received and would replace the symbol offset address table code section in the code storage section. An updated symbol accessor code section would be received and would replace the symbol accessor code section in the code storage section. Likewise, an updated patch manager code section (with a patch library) would be received and would replace the patch manager code section in the code storage section.

However, the above-mentioned code sections are typically bundled together in the patch manager code section. Thus, the read-write code section in the code storage section is replaced with the updated read-write code section from the file system section 110 when the patch manager code section 308 is replaced with the updated patch manger code section 450. Likewise, the code section address table, the symbol offset address table, the symbol accessor code sections, as well as the patch library are replaced when the updated patch manager code section 450 is installed. The arrangement of the new read-write data, the new code section address table, the new symbol offset address table, the new symbol accessor code, and the new patch library as the updated patch manager code section 450, together with the current code sections in the code storage section, forms updated executable system software.

When the file system section 110 receives an updated symbol accessor code address, the patch manager replaces the symbol accessor code address in the first location in memory with updated symbol accessor code address. As noted above, the first location in memory 334 is typically in the patch manager code section (see Fig. 3).

As seen in Fig. 3, the patch library 308 is also includes a compactor, or a compactor symbol library 342. The compactor 342 can also be enabled as a distinct and independent code section, however as noted above, it is useful and efficient to bundle the functions associated with system software upgrades into a single patch manager code section. Generally, the compactor 342 can be said to resize code sections, so that new sections can be arranged with current code sections in the code storage section 112.

Contrasting Figs. 3 and 4, the file system section 110 receives a compaction instruction set with instructions for identifying a current code section for updating via the airlink interface 150 (see Fig. 1). For example, the compaction instruction set can be the first PMRTI 454, although the instructions for compaction need not necessarily be bundled with the other updating instructions. The compactor 342 replaces the identified current code section in the code storage section with the new code section. It should be understood that the compactor 342 works in cooperation with the patch library 336 explained above. Alternately stated, the compactor 342 is called upon to assist the patch library 336 in circumstances where code section resizing is required in the process of updating the system software.

The compactor resizes the memory blocks (Fig. 4) in which corresponding resized code sections are stored in the code storage section. For example, the file system section 110 receives a new code section 450 having a first size. At the same time, or in other messages, the wireless device receives a compaction instruction set, say first PMRTI 454, and an updated patch manager code section 452. The compaction instruction set 454 identifies a current code section having a second size, say code section 2 (304), less than the first size. Since the updated code section 450 is larger in size that the code section being replaced, a compaction operation must be performed. The compactor 342 increases the size of the second memory block 402 associated with the identified current code section 304 to at least the first size.

The compactor (in cooperation with the patch library 336) replaces the identified current code section 304, stored in the corresponding memory block 402, with the new code section 450.

As noted above, it is common for the code sections to be oversized with respect to the symbol library, or symbol libraries arranged within. The compactor 342 determines the size of symbol libraries arranged within the corresponding code sections, and resizes code sections to more closely match the symbol library sizes arranged within. Typically, the compactor 342 optimally resizes code sections to further subsequent code section resizing and updating operations.

Continuing the example begun above, the compactor 342 might optimally resize code sections by taking unused areas from the code sections adjacent the second memory block 402. As shown, code section one (302), and a code section three (344) have unused areas 340 and 346, respectively. The compactor may choose to resize the first memory block 302 and a third memory block 460, taking from both unused areas 340 and 346 to make the second memory block 402 large enough for the new code section 450.

Fig. 8 is a schematic block diagram illustrating the results of exemplary compacting operation. Code section 2 has been replaced with the larger new (updated) code section 450. The unused section 340 of code section one (302) has been reduced, as has the unused area 360 of code section three (344). Leaving some unused areas in each code section makes for efficient future expansion and compaction operations. For example, if all the unused area 360 were used up to accommodate the updated code section 450 in the current update operation, any future expansion of code section three (344) would necessarily require compaction operations in adjacent code sections. It should also be understood that although the example only shows two adjacent memory blocks being resized, some compaction operations may require that some, or even all of the code sections be resized to accommodate new (larger) code section updates.

The compactor 342 accesses start addresses from code section address table, to measure the code sections sizes, and symbol offset addresses from the symbol offset address table, to measure the size of the symbol libraries arranged within corresponding code sections. Again, it should be understood that in some aspects of the invention that the compactor relies upon, and cooperates with other symbol libraries, that need not necessarily reside in the patch manager code section 308, to accomplish tasks. Preferably, however, the compactor works with the symbol accessor code described above to access the code section address table and symbol offset address table.

Since the code section address table includes the start address of each code section, a simple arithmetic operation using the start addresses of adjacent code sections can determine the size of the code section. Likewise, the address of the last symbol in a symbol library can be used, with the code section start addresses, to determine the approximate size of the symbol libraries arranged within corresponding code sections and, therefore, the amount of unused area. For example, if the start address of code section

one (302) is (0100), the start address of code section two (304) is (0200), and the address of the last symbol of symbol library one (310) is (0170), the size of unused area 340 is approximately (0200 - 0170).

However, many symbols are distributed over several contiguous addresses, and the offset address of the last symbol is not necessarily the same as the address where the symbol library ends. Continuing the above example, if the last symbol begins at address (0170) and ends at (0173), the size of the unused space 340 is actually (0200 - 0173). In some aspects of the invention, the code storage section 112 includes symbol libraries with end symbols to signify the end of symbol libraries. For example, returning momentarily to Fig. 6, symbol "Z" is a one-byte end symbol. That is, the address of symbol "Z" signifies the end of symbol library one. The compactor 342 uses the end symbol offset addresses, through the operation of the symbol accessor code to measure the size of symbol libraries arranged within corresponding code sections.

Alternately, the code storage section 112 includes symbol libraries with size symbols to signify the size of symbol libraries. For example, symbol "Y" (Fig. 6) can be a number (0073) that signifies the size of symbol library one. The compactor 342 accesses the size symbols to measure the size of symbol libraries arranged within corresponding code sections. In other aspects of the invention, the size symbols for all the symbol libraries are stored in a table in a distinct code section (not shown) of the patch manager code section.

The compactor 342 measures the size of the new code section 450 in the file system section 110 and first determines if the new code section can be arranged with the current code sections in the code storage section 112. The compactor 342 makes this determination in response to measuring the size of symbol libraries arranged within corresponding code sections, and measuring the size of the new code section. If compaction will not successfully permit the new code section to be installed, the compactor 342 automatically aborts the operation. Alternately, the compactor calculates a probability of success, displays the probability to the wireless device user, and permits the user to determine if the compaction operation should continue. Regardless of the probability of success, the wireless telephone gives the user the option of starting the compaction process, as the compactor 342 must suspend the execution of the system software during the compaction and updating process.

To determine the size of the updated code section, the compactor 342 generally relies upon compaction instruction set data. The file system section 110 receives a compaction instruction set 454 that includes the size of the new code section 450. The compactor 342 accesses the compaction instruction set 454 to determine the size of the new code section 450 in the file system section 110 and determines if the new code section can be arranged with the current code sections in the code storage section 112, in response to measuring the size of symbol libraries arranged within corresponding code sections and determining the size of the new code section.

The compaction operation can be avoided if the code storage section includes large unused memory block sections. The compactor 342 determines the size of unused memory blocks in the code storage section and stores the new code section in the unused memory block, if the size of the unused memory block is greater than, or equal to the new code section size.

Once the code sections have been resized, and the identified code sections replaced with updated code sections, the compactor 342 changes the start addresses of code sections stored in the code storage section. Continuing the above example, if the resizing operation is accomplished by starting the new code section 450 (now code section two) at the start address of (0175), instead of (0200), the code section address table must be changed to reflect the new start address.

Figs. 9a through 9e illustrate the operation of an exemplary compaction schedule. After the compactor 342, operating from volatile memory 114, calculates the probability of successful compaction, and calculates the code section sizes, the compactor 342 generates a compaction schedule. The compaction schedule includes the order in which code sections are moved, and their temporary placement in the FSS 110. The file system section 110 temporarily stores code sections from the code storage section 112, and the compactor 342 stores the code sections from the file system section 110 into the code storage section 112 memory blocks to maintain contiguous addressing, in response to the compaction schedule.

Continuing the example begun above, in Fig. 9a the compactor 342 consults the compaction instruction set (first PMRTI) 454 to determine the code section to be updated and the size of the new code section 450. In Fig. 9b, the compactor 342, after calculating the probability of success and size of the code sections, begins the process by moving code section three (344) to FSS 110 from the third memory block 460.

In Fig. 9c, the new code section 450 is moved from FSS 110 to the second memory block 402. If code section one (302) has been resized, the compactor 342 causes the start address of the new code section to begin in the unused area 340 (see Fig. 3). As mentioned above, the start of the second memory block changes from (0200) to (0175). That is, code section one (302) has been compacted.

In Fig. 9d, code section three (344) is moved back from FSS 110 to the third memory block 460. The move is made to start code section three (344) at a new start address. Starting the code section at a new start address may involve compacting the code section. That is, some of the unused area 346 may be used.

In Fig. 9e, the patch manager (PM) code section 308 is replaced with the new (updated) PM code section 452. Typically, the patch manager includes the code section address table code symbol library (see Fig. 3). However, the updated code section address table is not necessarily correct, as the results of compaction are not necessarily known beforehand. Then, the compactor 342 generates an updated code section address table, in response to changing the start addresses of code sections stored within the memory blocks. The compactor 342 overwrites the updated code section address table stored in the code

storage section 112 with the updated code section address table that it has generated. Then, the updated system software accesses the updated code section address table after the new code section is arranged with the current code sections.

However, the symbol offset address table is handled differently, since the symbol offset addresses for updated and current code sections can be calculated beforehand. The file system section 110 receives a new code section, typically the PM code section 452, with an updated symbol offset address table, and the updated system software accesses the updated symbol offset address table after the new code section 452 is arranged with the current code sections.

Much of the above description of the invention has been devoted to the operation of the compactor in calculating code section sizes and a compaction schedule. However, in some aspects of the invention the compactor 342 is relieved of the responsibility of these calculations. In this aspect of the invention, the file system section 110 receives a compaction instruction 454 set including code section resizing instructions and a compaction schedule. Then, the compactor 342 resizes code sections in response to the code section resizing instructions. As before, the file system section 110 temporarily stores code sections from the code storage section 112, and the compactor 342 stores the code sections from the file system section 110 into the code storage section 112 memory blocks to maintain contiguous addressing, in response to the compaction schedule.

The file system section 110 also receives a new code section, for example the updated PM code section 452, with an updated code section address table and an updated symbol offset address table. Since the code section resizing and compaction schedule were calculated beforehand (not by the wireless device), these updated tables accurately reflect the new code section start addresses. The updated system software accesses the updated symbol offset address table and updated code section address table after the new code section is arranged with the current code sections.

As best illustrated in Fig. 1, the volatile memory 114 includes the first and second table code sections (the code section address table and symbol offset address table) loaded from the code storage section 112 for executing the system software. Other critical libraries from the patch manager code section 308 are also loaded into volatile memory 114. The new code sections are arranged with the current code sections in the code storage section 112 to form updated system software, following a reset of the wireless communications device 104. When reset, the updated code section address table and updated symbol offset address table are loaded into the volatile memory 114, and the system software can then be executed with the updated tables.

Figs. 10a through 10b are a flowchart illustrating the present invention method for reorganizing software instructions stored in a wireless communications device memory. Although depicted as a sequence of numbered steps for clarity, no order should be inferred from the numbering unless explicitly stated. The method begins at Step 1000. Step 1001a forms the system software into a first plurality of

symbol libraries, each symbol library comprising at least one symbol. Step 1001b arranges the first plurality of symbol libraries into a second plurality of code sections. Step 1002 stores wireless device system software in a plurality of current code sections. Step 1004 receives a new code section. Receiving a new code section in Step 1004 includes receiving the new code section via a wireless communications device air interface.

Step 1006 resizes current code sections. Typically, resizing current code sections includes suspending the operation of the system software. Step 1008 arranges the new code section with the current code sections to form updated system software for the wireless device. Step 1010 executes the updated system software. Typically, Step 1005a identifies a current code section for updating, and arranging the new code section with the current code sections to form updated system software in Step 1008 includes replacing the identified current code section with the new code section.

Arranging the first plurality of symbol libraries into a second plurality of code sections in Step 1001b includes starting symbol libraries at the start of code sections. Storing wireless device system software in a plurality of current code sections in Step 1002 includes storing the start of code sections at corresponding start addresses. Then, the method comprises a further step. Step 1003a maintains a code section address table cross-referencing code section identifiers with corresponding start addresses. Arranging the first plurality of symbol libraries into a second plurality of code sections in Step 1001b includes arranging symbols to be offset from their respective code section start addresses. Then, Step 1003b maintains a symbol offset address table cross-referencing symbol identifiers with corresponding offset addresses, and corresponding code section identifiers.

Storing the start of code sections at corresponding start addresses includes substeps. Step 1002a creates a second plurality of contiguously addressed memory blocks. Step 1002b identifies each memory block with a corresponding code section. Step 1002c stores code sections in the identified memory blocks.

Arranging the first plurality of symbol libraries into a second plurality of code sections in Step 1002 includes sizing the code sections to accommodate arranged symbol libraries. Creating a second plurality of contiguously addressed memory blocks in Step 1002a includes sizing memory blocks to accommodate corresponding code sections. Typically, sizing memory blocks to accommodate corresponding code sections includes sizing the code sections to accommodate sizes larger than the arranged symbol libraries. Resizing current code sections in Step 1006 includes resizing the memory blocks in which corresponding resized code sections are stored.

For example, receiving a new code section in Step 1004 includes receiving a new code section having a first size. Identifying a current code section for updating in Step 1005a includes identifying a current code section having a second size, less than the first size. Then, resizing the memory blocks in which corresponding resized code sections are stored in Step 1006 includes substeps. Step 1006a (not

shown) increases the size of a memory block associated with the identified current code section to at least the first size, and Step 1006b (not shown) replaces the identified current code section, stored in the corresponding memory block, with the new code section.

Resizing the memory blocks in which corresponding resized code sections are stored in Step 1006 includes, in response to measuring the size of the symbol libraries arranged within corresponding code sections (see Step 1005c, below), resizing code sections to more closely match the symbol library sizes arranged within. Resizing code sections to more closely match the symbol library sizes arranged within in Step 1006 includes optimally resizing code sections to further subsequent code section resizing and updating operations.

Step 1005b, using the start addresses from code section address table, measures the current code sections sizes. Step 1005c, using the symbol offset addresses from the symbol offset address table, measures the size of the symbol libraries arranged within corresponding code sections. That is, the code section sizes and sizes of the symbol libraries are measured. This information permits the amount of unused area in each code section to be calculated.

In some aspects, forming the system software into a first plurality of symbol libraries in Step 1001a includes forming end symbols to signify the end of symbol libraries. Then, measuring the size of the symbol libraries arranged within corresponding code sections in Step 1005c includes using the offset addresses of the end symbols to measure the size of symbol libraries. Alternately, forming the system software into a first plurality of symbol libraries in Step 1001a includes forming size symbols to signify the size of symbol libraries. Then, measuring the size of the symbol libraries arranged within corresponding code sections in Step 1005c includes accessing the size symbols to measure the size of symbol libraries.

Step 1005d measures the size of the new code section. Step 1005e, in response to measuring the size of symbol libraries arranged within corresponding code sections in Step 1005c, and measuring the size of the new code section in Step 1005d, determines if the new code section can be arranged with the current code sections.

In some aspects, determining if the new code section can be arranged with the current code section in Step 1005e includes determining the size of unused memory blocks. Then, arranging the new code section with the current code sections to form updated system software for the wireless device in Step 1008 includes storing the new code section in the unused memory block, if the size of the unused memory block is greater than, or equal to the new code section size.

Step 1005f calculates the code section sizes. That is, a determination is made of new code sections sizes that will permit the new code section to be arranged with the current code sections. Step 1007a, in response to calculating the code section sizes, generates a compaction schedule. Step 1007b temporarily moves code sections into a file system section. Then, arranging the new code section with the

current code sections to form updated system software for the wireless device in Step 1008 includes storing the code sections from the file system section into the memory blocks to maintain contiguous addressing, in response to the compaction schedule.

Following the resizing of the code sections in Step 1006, Step 1009a changes the code section start addresses. Step 1009b, in response to changing the start addresses of code sections stored within the memory blocks, generates an updated code section address table. Then, executing the updated system software in Step 1010 includes using the updated code section address table after arranging the new code section with the current code sections.

In some aspects of the invention receiving a new code section in Step 1004 includes receiving a new code section with an updated symbol offset address table. Then, executing the updated system software in Step 1010 includes using the updated symbol offset address table after arranging the new code section with the current code sections.

Step 1003c loads the code section address table and symbol offset address table into a volatile memory. Step 1003d (not shown), in response to loading the code section address table and symbol offset address table into the volatile memory, executes system software. Step 1009c resets the wireless communications device. Step 1009d, in response to resetting, loads the updated code section address table and the updated symbol offset address table into volatile memory. Step 1010, in response to loading the updated code section address table and updated symbol offset address table into memory, executes the updated system software.

Fig. 11 is a flowchart illustrating an alternate aspect of the method of Figs. 10a through 10b. Fig. 11 shares several steps in common with Figs. 10a and 10b, which for the sake of brevity are repeated here. Receiving a new code section in Step 1104 includes receiving a compaction instruction set including code section resizing instructions and a compaction schedule. Resizing current code sections in Step 1106 includes resizing in response to the code section resizing instructions.

Step 1107 temporarily moves code sections into a file system section. Then, arranging the new code section with the current code sections to form updated system software for the wireless device in Step 1008 includes storing the code sections from the file system section into memory blocks to maintain contiguous addressing, in response to the compaction schedule.

Receiving a new code section in Step 1104 includes receiving a new code section with an updated code section address table and an updated symbol offset address table. Then, executing the updated system software in Step 1110 includes using the updated code section address table and updated symbol offset address table after arranging the new code section with the current code sections. A system and method have been provided for reorganizing system software structure in a wireless communications device so aid in the process of updating the software. The system is easily updateable because of the arrangement of symbol libraries in code sections, with tables to access the start addresses of the code

sections in memory and the offset addresses of symbols in the symbol libraries. Although a few examples of these library arrangements and cross-referencing tables have been given for a display function, the present invention is not limited to just these examples. Other variations and embodiments of the invention will occur to those skilled in the art.

CLAIMS

WHAT IS CLAIMED IS:

1. In a wireless communications device, a method for reorganizing software instructions stored in a memory, the method comprising:
storing wireless device system software in a plurality of current code sections;
receiving a new code section;
resizing current code sections;
arranging the new code section with the current code sections to form updated system software for the wireless device; and,
executing the updated system software.

2. The method of claim 1 further comprising:
identifying a current code section for updating; and,
wherein arranging the new code section with the current code sections to form updated system software includes replacing the identified current code section with the new code section.

3. The method of claim 2 further comprising:
forming the system software into a first plurality of symbol libraries, each symbol library comprising at least one symbol; and,
arranging the first plurality of symbol libraries into a second plurality of code sections.

4. The method of claim 3 wherein receiving a new code section includes receiving the new code section via a wireless communications device air interface.

5. The method of claim 4 wherein arranging the first plurality of symbol libraries into a second plurality of code sections includes starting symbol libraries at the start of code sections;
wherein storing wireless device system software in a plurality of current code sections includes storing the start of code sections at corresponding start addresses; and,
the method further comprising:
maintaining a code section address table cross-referencing code section identifiers with corresponding start addresses.

6. The method of claim 5 wherein arranging the first plurality of symbol libraries into a second plurality of code sections includes arranging symbols to be offset from their respective code section start addresses; and

the method further comprising:
maintaining a symbol offset address table cross-referencing symbol identifiers with corresponding offset addresses, and corresponding code section identifiers.

7. The method of claim 6 wherein storing the start of code sections at corresponding start addresses includes:
creating a second plurality of contiguously addressed memory blocks;
identifying each memory block with a corresponding code section; and,
storing code sections in the identified memory blocks.

8. The method of claim 7 wherein arranging the first plurality of symbol libraries into a second plurality of code sections includes sizing the code sections to accommodate arranged symbol libraries; and,
wherein creating a second plurality of contiguously addressed memory blocks includes sizing memory blocks to accommodate corresponding code sections.

9. The method of claim 8 wherein sizing memory blocks to accommodate corresponding code sections includes sizing the code sections to accommodate sizes larger than the arranged symbol libraries.

10. The method of claim 9 wherein resizing current code sections includes resizing the memory blocks in which corresponding resized code sections are stored.

11. The method of claim 10 wherein receiving a new code section includes receiving a new code section having a first size;
wherein identifying a current code section for updating includes identifying a current code section having a second size, less than the first size; and,
wherein resizing the memory blocks in which corresponding resized code sections are stored includes:
increasing the size of a memory block associated with the identified current code section to at least the first size; and,
replacing the identified current code section, stored in the corresponding memory block, with the new code section.

12. The method of claim 11 wherein resizing the memory blocks in which corresponding resized code sections are stored includes, in response to measuring the size of symbol libraries arranged within the

corresponding code sections, resizing code sections to more closely match the symbol library sizes arranged within.

13. The method of claim 12 wherein resizing code sections to more closely match the symbol library sizes arranged within includes optimally resizing code sections to further subsequent code section resizing and updating operations.

14. The method of claim 12 further comprising:
using the start addresses from code section address table, measuring the current code sections sizes;
using the symbol offset addresses from the symbol offset address table, measuring the size of the symbol libraries arranged within corresponding code sections.

15. The method of claim 14 wherein forming the system software into a first plurality of symbol libraries includes forming end symbols to signify the end of symbol libraries; and,
wherein measuring the size of the symbol libraries arranged within corresponding code sections includes using the offset addresses of the end symbols to measure the size of symbol libraries.

16. The method of claim 14 wherein forming the system software into a first plurality of symbol libraries includes forming size symbols to signify the size of symbol libraries; and,
wherein measuring the size of the symbol libraries arranged within corresponding code sections includes accessing the size symbols to measure the size of symbol libraries.

17. The method of claim 15 further comprising:
following the resizing of the code sections, changing the code section start addresses.

18. The method of claim 16 further comprising:
measuring the size of the new code section; and,
in response to measuring the size of symbol libraries arranged within corresponding code sections, and measuring the size of the new code section, determining if the new code section can be arranged with the current code sections.

19. The method of claim 18 wherein determining if the new code section can be arranged with the current code section includes determining the size of unused memory blocks; and,

wherein arranging the new code section with the current code sections to form updated system software for the wireless device includes storing the new code section in the unused memory block, if the size of the unused memory block is greater than, or equal to the new code section size.

20. The method of claim 18 further comprising:
calculating the code section sizes;
in response to calculating the code section sizes, generating a compaction schedule;
temporarily moving code sections into a file system section; and,
wherein arranging the new code section with the current code sections to form updated system software for the wireless device includes storing the code sections from the file system section into the memory blocks to maintain contiguous addressing, in response to the compaction schedule.

21. The method of claim 20 further comprising:
in response to changing the start addresses of code sections stored within the memory blocks, generating an updated code section address table; and,
wherein executing the updated system software includes using the updated code section address table after arranging the new code section with the current code sections.

22. The method of claim 21 wherein receiving a new code section includes receiving a new code section with an updated symbol offset address table; and,
wherein executing the updated system software includes using the updated symbol offset address table after arranging the new code section with the current code sections.

23. The method of claim 11 wherein receiving a new code section includes receiving a compaction instruction set including code section resizing instructions and a compaction schedule; and,
wherein resizing current code sections includes resizing code sections in response to the code section resizing instructions.

24. The method of claim 23 further comprising:
temporarily moving code sections into a file system section; and,
wherein arranging the new code section with the current code sections to form updated system software for the wireless device includes storing the code sections from the file system section into memory blocks to maintain contiguous addressing, in response to the compaction schedule.

25. The method of claim 24 wherein receiving a new code section includes receiving a new code section with an updated code section address table and an updated symbol offset address table; and, wherein executing the updated system software includes using the updated code section address table and updated symbol offset address table after arranging the new code section with the current code sections.

26. The method of claim 6 further comprising:
loading the code section address table and symbol offset address table into a volatile memory;
in response to loading the code section address table and symbol offset address table into the volatile memory, executing system software;
resetting the wireless communications device;
in response to resetting, loading the updated code section address table and the updated symbol offset address table into volatile memory; and,
wherein executing the updated system software includes executing the updated system software in response to loading the updated code section address table and updated symbol offset address table into memory.

27. The system of claim 26 wherein resizing current code sections includes suspending the operation of the system software.

28. In a wireless communications device, a method for reorganizing software instructions stored in a memory, the method comprising:
storing wireless device system software in a plurality of current code sections with the start of code sections at corresponding start addresses by creating a second plurality of contiguously addressed memory blocks, identifying each memory block with a corresponding code section, and storing code sections in identified memory blocks;
receiving a new code section via a wireless communications device air interface;
identifying a current code section for updating;
calculating the code section sizes;
in response to calculating the code section sizes, generating a compaction schedule;
resizing current code sections;
following the resizing of the current code sections, changing the code section start addresses;
temporarily moving code sections into a file system section;
replacing the identified current code section with the new code section by storing the code sections from the file system section into memory blocks to maintain contiguous addressing, in response to the compaction schedule; and,

executing the updated system software.

29. In a wireless communications device, a system for reorganizing software instructions stored in a memory, the system comprising:
a code storage section memory including executable wireless device system software differentiated into a plurality of current code sections;
a file system section memory for receiving new code sections;
a compactor to resize current code sections; and
wherein the arrangement of new code sections with the current code sections in the code storage section forms updated system software.

30. The system of claim 29 wherein the file system section receives a compaction instruction set with instructions for identifying a current code section for updating; and,
wherein the compactor replaces the identified current code section in the code storage section with the new code section.

31. The system of claim 30 wherein the code storage section comprises a first plurality of symbol libraries, each symbol library comprising at least one symbol, with the first plurality of symbol libraries being arranged into a second plurality of code sections.

32. The system of claim 31 further comprising:
an airlink interface to receive new code sections; and,
wherein the file system section receives the new code section via a wireless communications device airlink interface.

33. The system of claim 32 wherein the code storage section symbol libraries start at the start of code sections; and,
wherein the code storage section includes a first table code section with a code section address table for cross-referencing code section identifiers with corresponding start addresses.

34. The system of claim 33 wherein the code storage section includes symbols arranged to be offset from their respective code section start addresses; and,
wherein the code storage section includes a second table code section with a symbol offset address table for cross-referencing symbol identifiers with corresponding offset addresses, and corresponding code section identifiers.

35. The system of claim 34 wherein the code storage section includes a second plurality of contiguously addressed memory blocks identified with the corresponding second plurality of code sections.

36. The system of claim 35 wherein the code storage section includes code sections sized to accommodate the symbol libraries arranged within, and memory blocks sized to accommodate the corresponding code sections.

37. The system of claim 36 wherein the code storage section includes code sections sized to accommodate sizes larger than the symbol libraries arranged within.

38. The system of claim 37 wherein the compactor resizes the memory blocks in which corresponding resized code sections are stored in the code storage section.

39. The system of claim 38 wherein the file system section receives a new code section having a first size;
wherein compaction instruction set identifies a current code section having a second size, less than the first size; and,
wherein the compactor increases the size of a memory block associated with the identified current code section to at least the first size, and replaces the identified current code section, stored in the corresponding memory block, with the new code section.

40. The system of claim 39 wherein the compactor determines the size of symbol libraries arranged within the corresponding code sections, and resizes code sections to more closely match the symbol library sizes arranged within.

41. The system of claim 40 wherein the compactor optimally resizes code sections to further subsequent code section resizing and updating operations.

42. The system of claim 40 wherein the compactor accesses start addresses from code section address table, to measure the code sections sizes; and,
wherein the compactor accesses symbol offset addresses from the symbol offset address table, to measure the size of the symbol libraries arranged within corresponding code sections.

43. The system of claim 42 wherein the code storage section includes symbol libraries with end symbols to signify the end of symbol libraries; and,
wherein the compactor uses the end symbol offset addresses to measure the size of symbol libraries arranged within corresponding code sections.

44. The system of claim 42 wherein the code storage section includes symbol libraries with size symbols to signify the size of symbol libraries; and,
wherein the compactor accesses the size symbols to measure the size of symbol libraries arranged within corresponding code sections.

45. The system of claim 43 wherein the compactor changes the start addresses of code sections stored in the code storage section, after resizing the code sections.

46. The system of claim 45 wherein the compactor measures the size of the new code section in the file system section and determines if the new code section can be arranged with the current code sections in the code storage section, in response to measuring the size of symbol libraries arranged within corresponding code sections and measuring the size of the new code section.

47. The system of claim 45 wherein the file system section receives an compaction instruction set including the size of the new code section; and,
wherein the compactor accesses the compaction instruction set to determine the size of the new code section in the file system section and determines if the new code section can be arranged with the current code sections in the code storage section, in response to measuring the size of symbol libraries arranged within corresponding code sections and determining the size of the new code section.

48. The system of claim 45 wherein the compactor determines the size of unused memory blocks in the code storage section and stores the new code section in the unused memory block, if the size of the unused memory block is greater than, or equal to the new code section size.

49. The system of claim 45 wherein the compactor calculates the code section sizes and, in response to calculating the code section sizes, generates a compaction schedule;
wherein the file system section temporarily stores code sections from the code storage section; and,
wherein the compactor stores the code sections from the file system section into the code storage section memory blocks to maintain contiguous addressing, in response to the compaction schedule.

50. The system of claim 49 wherein the compactor generates an updated code section address table, in response to changing the start addresses of code sections stored within the memory blocks, and,
wherein the updated system software accesses the updated code section address table after the new code section is arranged with the current code sections.

51. The system of claim 50 wherein the file system section receives a new code section with an updated symbol offset address table; and,
wherein the updated system software accesses the updated symbol offset address table after the new code section is arranged with the current code sections.

52. The system of claim 39 wherein the file system section receives a compaction instruction set including code section resizing instructions and a compaction schedule; and,
wherein the compactor resizes code sections in response to the code section resizing instructions.

53. The system of claim 52 wherein the file system section temporarily stores code sections from the code storage section; and,
wherein the compactor stores the code sections from the file system section into the code storage section memory blocks to maintain contiguous addressing, in response to the compaction schedule.

54. The system of claim 53 wherein the file system section receives a new code section with an updated code section address table and an updated symbol offset address table; and,
wherein the updated system software accesses the updated symbol offset address table and updated code section address table after the new code section is arranged with the current code sections.

55. The system of claim 29 wherein the code storage section and file system section memories are nonvolatile memories.

56. The system of claim 34 further comprising:
a volatile memory including the first and second table code sections loaded from the code storage section for executing the system software; and,
wherein the arrangement of new code sections with the current code sections in the code storage section forms updated system software, following a reset of the wireless communications device to load the updated code section address table and updated symbol offset address table in the volatile memory.

57. The system of claim 56 wherein the compactor suspends the execution of the system software.

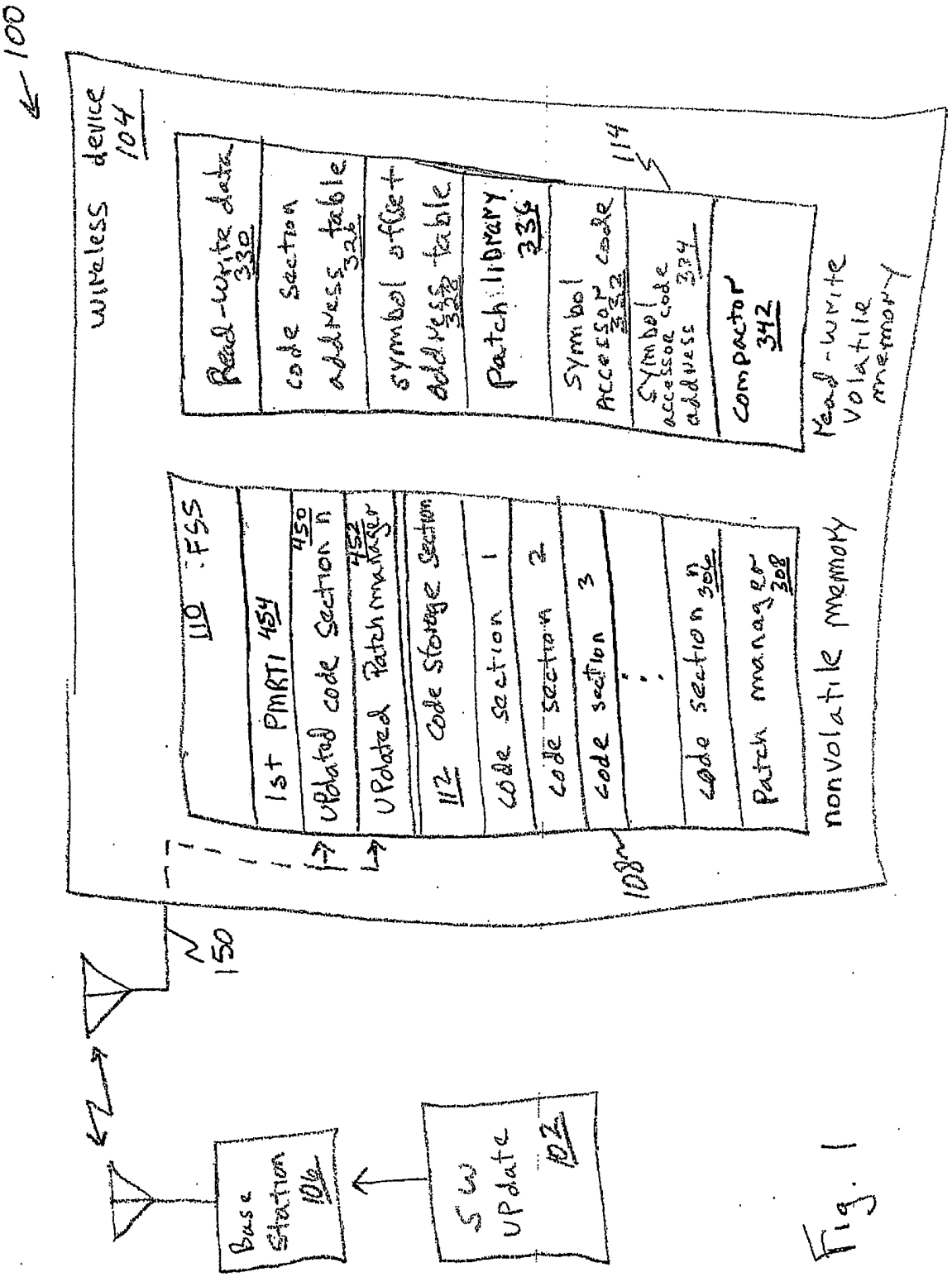
58. In a wireless communications device, a system for reorganizing software instructions stored in a memory, the system comprising:

a code storage section memory including executable wireless device system software differentiated into a plurality of current code sections with the start addresses identified with a plurality of contiguously addressed memory blocks;

a file system section for receiving new code sections, via an airlink interface, including a compaction instruction set identifying the current code section for updating;

a compactor to calculate the code section sizes, generate a compaction schedule, resize current code sections, temporarily move code sections into a file system section and replace the identified current code section with the new code section by storing the code sections into memory blocks to maintain contiguous addressing; and,

wherein the arrangement of new code sections with the current code sections in the code storage section forms updated system software.


100
Wireless device 104
Read-write data 330
code section address table 326
symbol offset address table 328
Patch library 336
Symbol accessor code 332
Symbol accessor address 334
Compactor 342
114
Read-write Volatile memory
I/O FSS
1st PMRT 454
450
Updated code section n
452
Updated Patch manager
112 Code Storage Section
Code section 1
Code section 2
Code section 3
Code section n 306
Patch manager 308
108
nonvolatile memory
150
Base station 102a
SW Update 102
Fig. 1

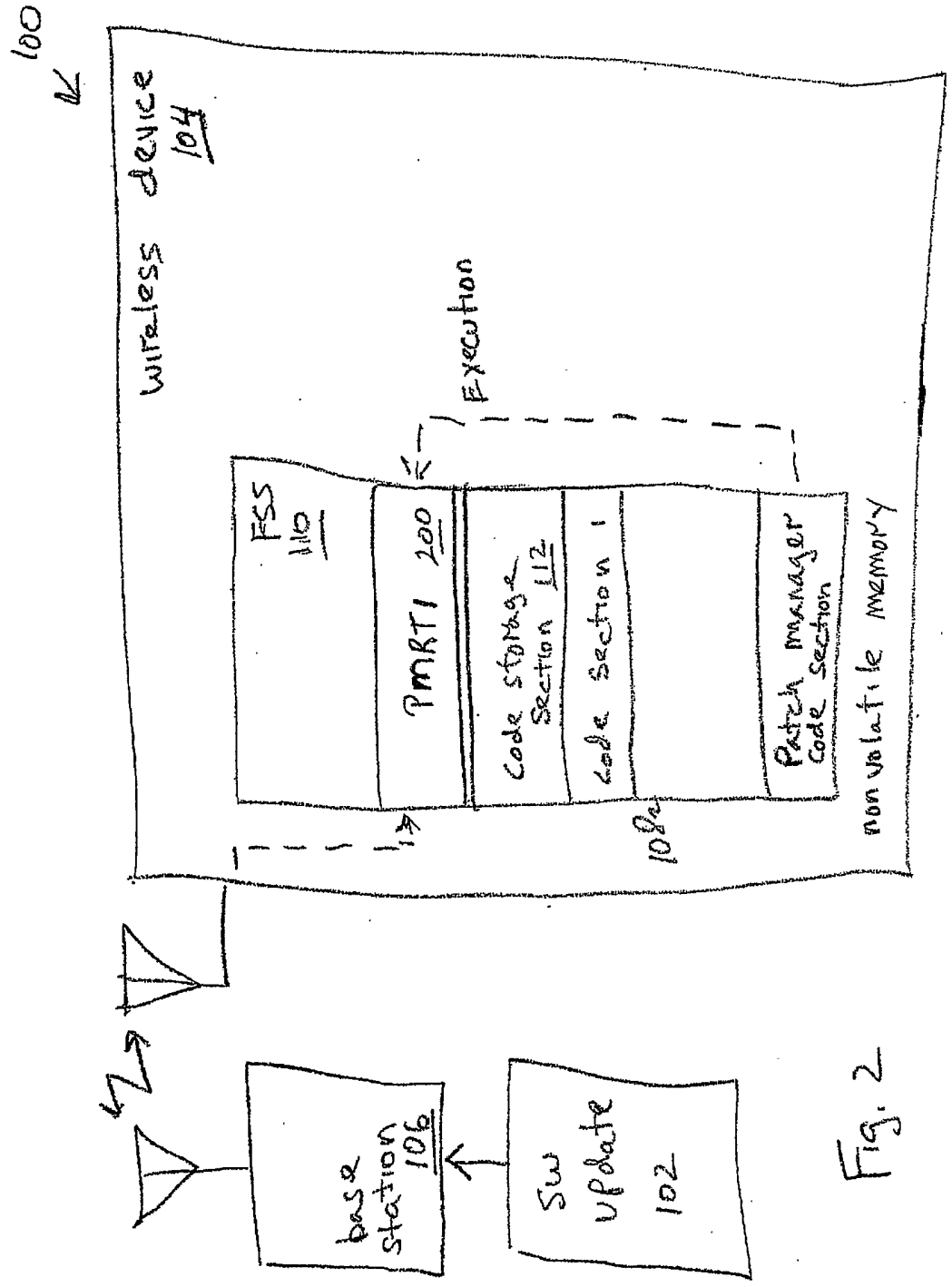

100
wireless device 104
FSS 110
PMRTI 200
Code storage Section 112
Code Section 1
Patch manager Code section
non volatile memory
108a
Execution
base station 106
SW Update 102

Fig. 2

3116

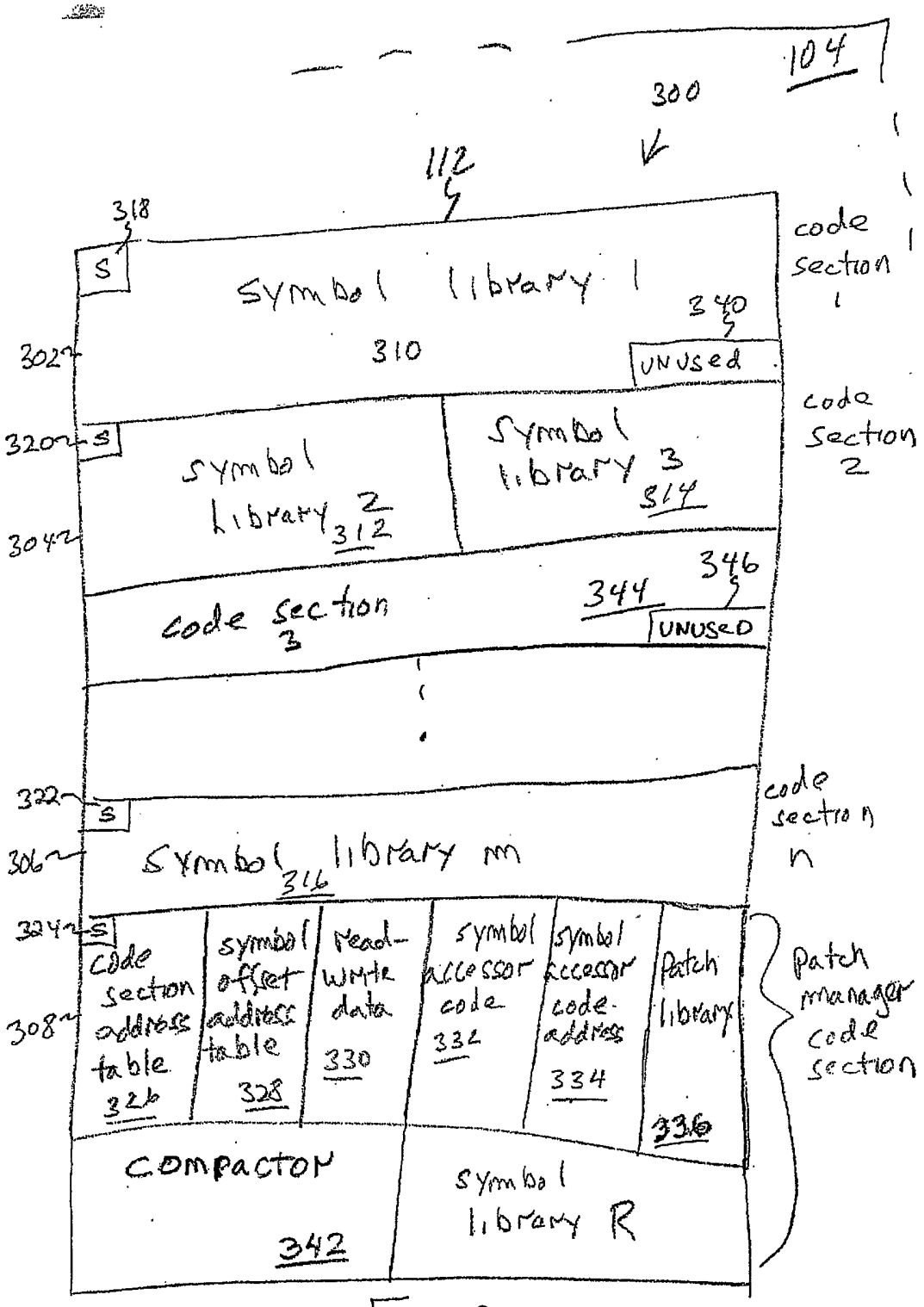

104
300
112
318
S
Symbol library 1
310
340
UNUSED
code section 1
302
320
S
symbol library 2
312
Symbol library 3
314
code section 2
304
code section 3
344
346
UNUSED
322
S
code section n
306
Symbol library m
316
324
S
code section address table
326
symbol offset address table
328
read-write data
330
symbol accessor code
332
symbol accessor code address
334
patch library
336
patch manager code section
308
COMPACTOR
342
Symbol library R

Fig. 3

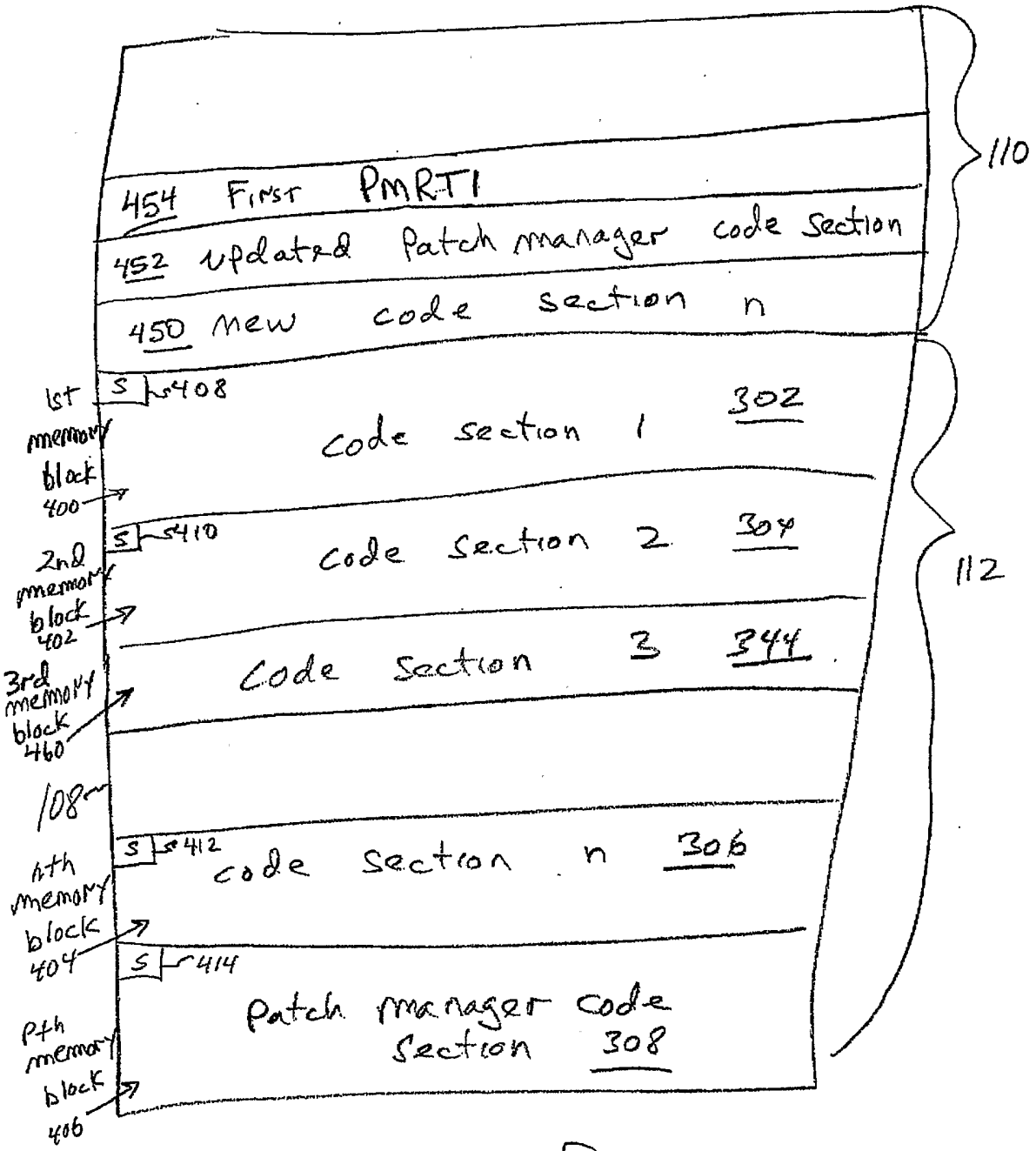

454 First PMRT1
452 updated Patch manager code section
450 new code section n
110
1st memory block 400
S 408
code section 1 302
2nd memory block 402
S 410
code section 2 304
112
3rd memory block 460
code section 3 344
108
Nth memory block 404
S 412
code section n 306
Pth memory block 406
S 414
Patch manager code section 308

Fig. 4

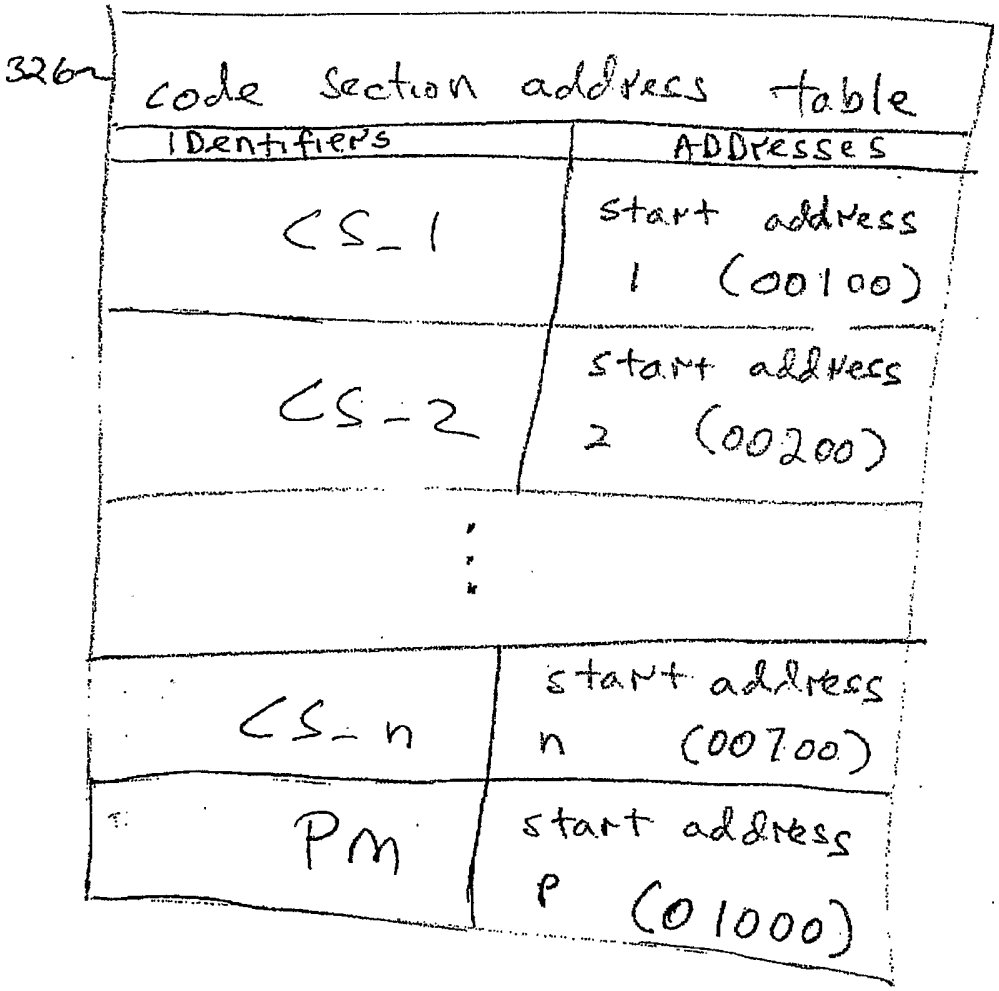

326
code section address table
IDentifiers
ADDresses
CS_1
start address 1 (00100)
CS_2
start address 2 (00200)
CS_n
start address n (00700)
PM
start address p (01000)

Fig. 5

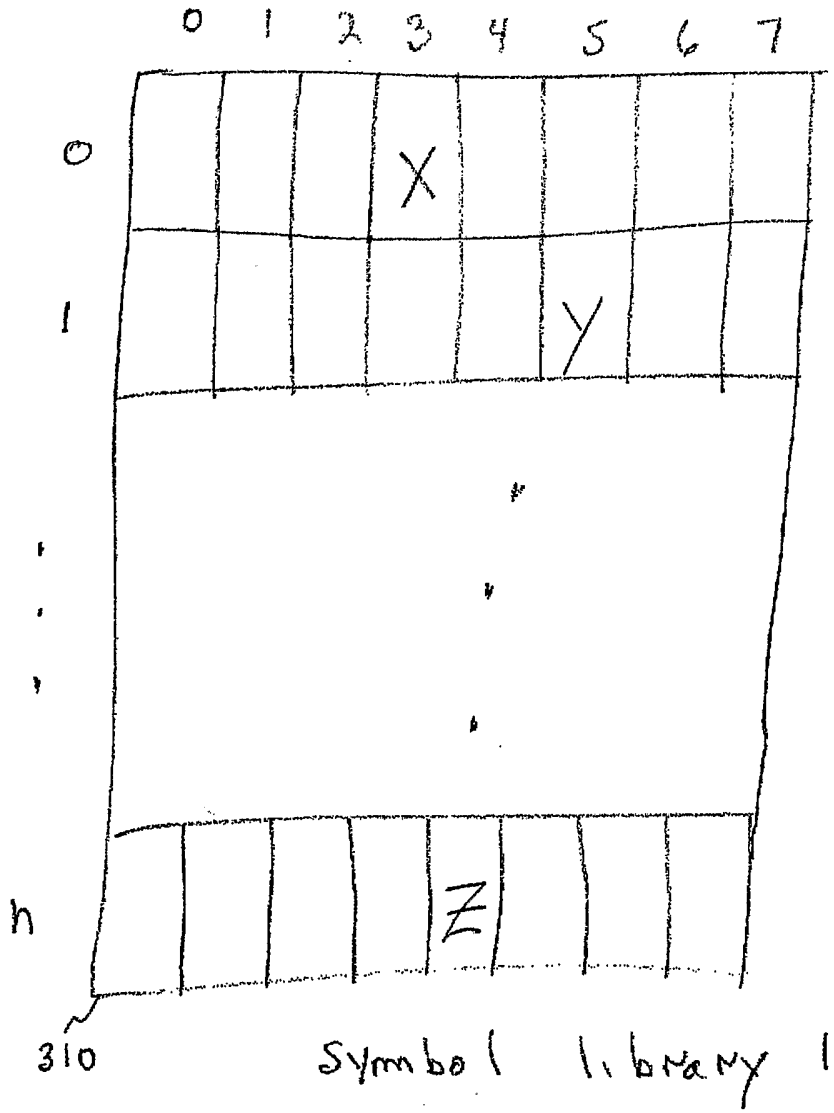


Fig. 6

Symbol offset address table

Symbol ID	code section ID	offset
X_1	CS_1	03
Y_1	CS_1	15
P_1	CS_2	11
Q_1	CS_2	33
AA_3	CS_2	47
⋮		

Fig. 7

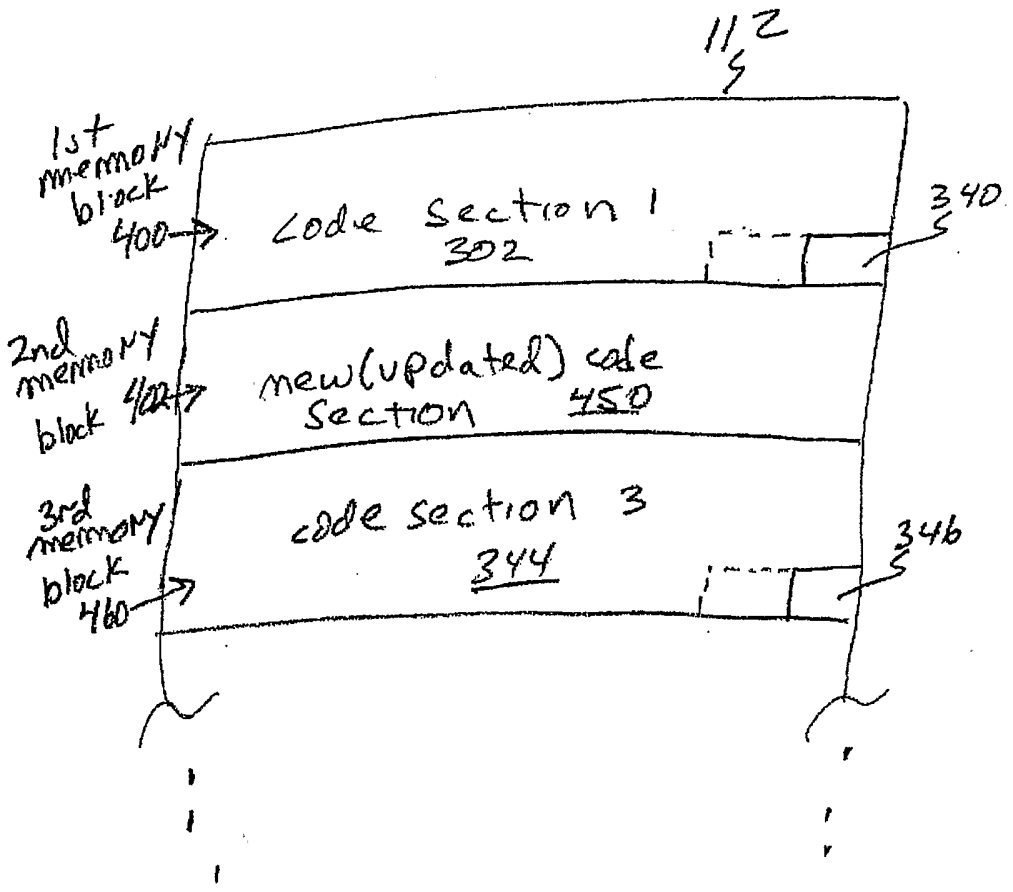

112
1st memory block 400
code section 1 302
340
2nd memory block 402
new(updated) code section 450
3rd memory block 460
code section 3 344
346

Fig. 8

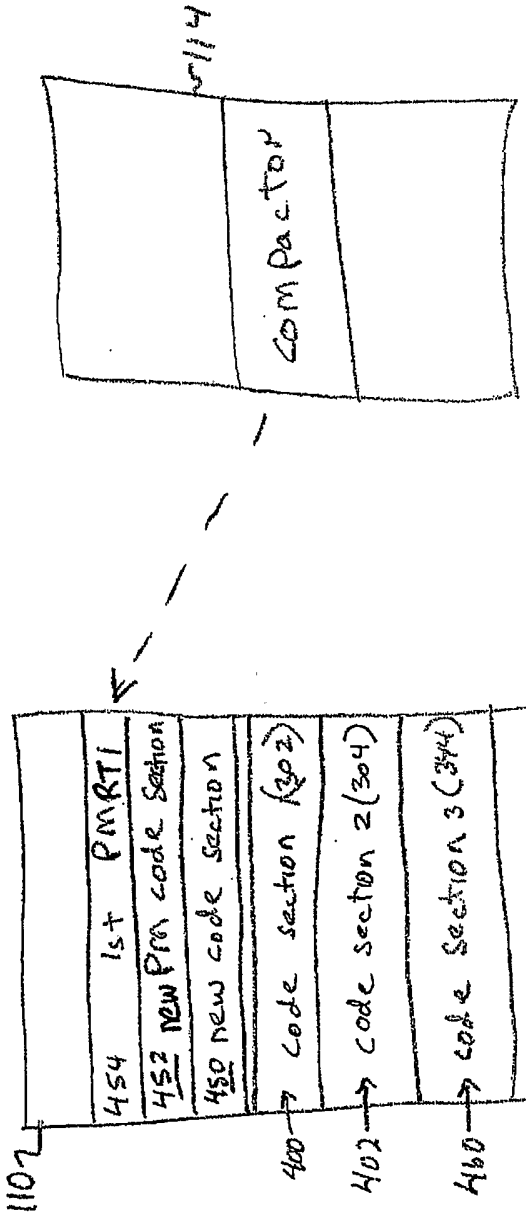

114
Compactor
110
454
1st PMRTI
452 new PM code Section
450 new code section
400 code section 1 (302)
402 code section 2 (304)
460 code section 3 (344)
112 PM code section 308

Fig. 9a

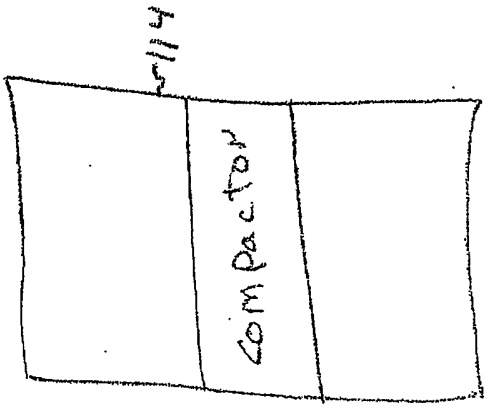
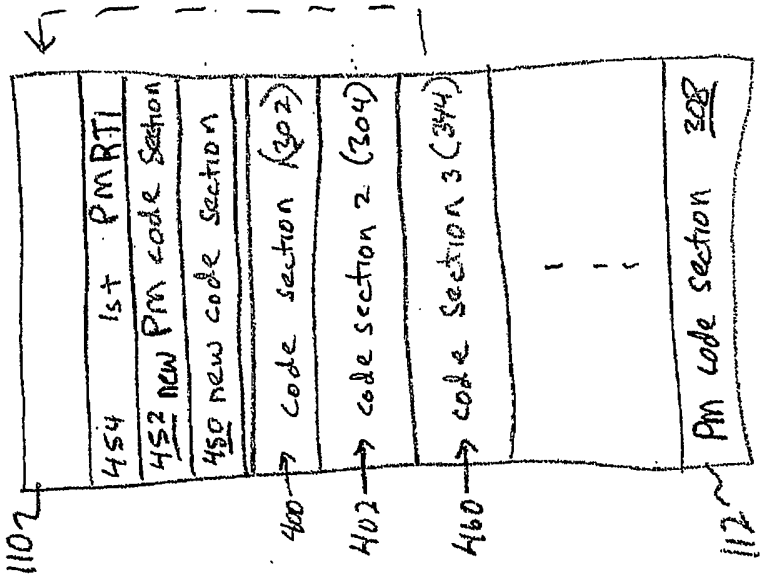

114
compactor

Fig. 9b


110
454 1st PMRTI
452 new PM code section
450 new code section
400
code section 1 (302)
402
code section 2 (304)
460
code section 3 (344)
308
PM code section
112

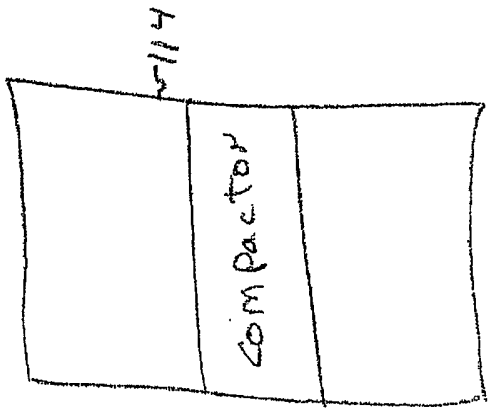
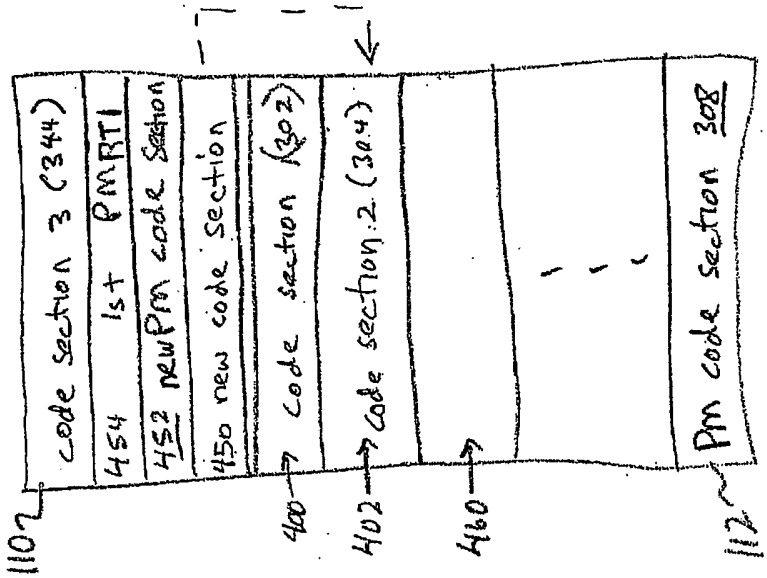

114
Compactor

Fig. 9c


code section 3 (344)
454 1st PMRTI
452 newPm code Section
450 new code section
code section (302)
code section 2 (304)
Pm code section 308
110
400
402
460
112

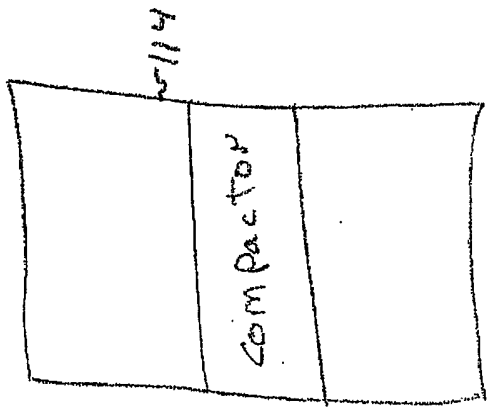
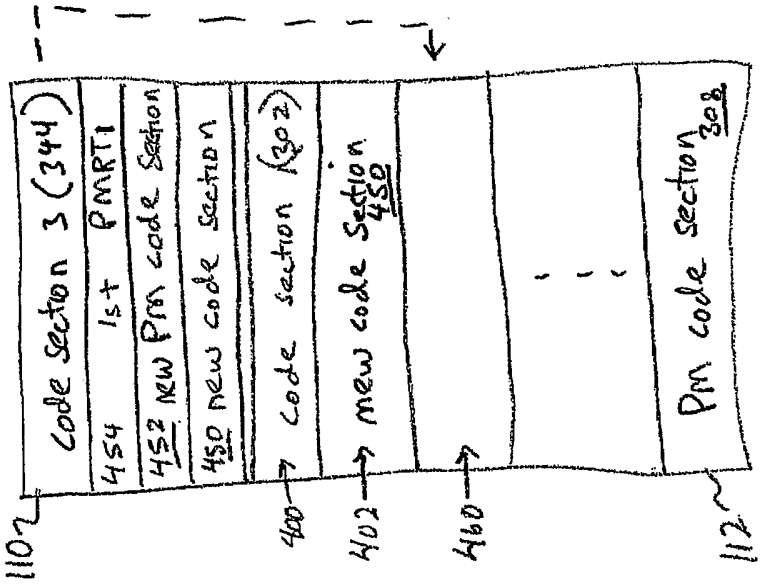

114
Compactor

Fig. 9d


Code section 3 (344)
454 1st PMRTI
452 new PM code section
450 new code section
Code section (302)
new code section 450
PM code section 308
110
400
402
460
112

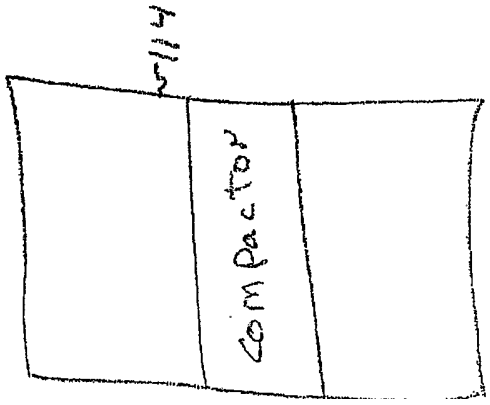
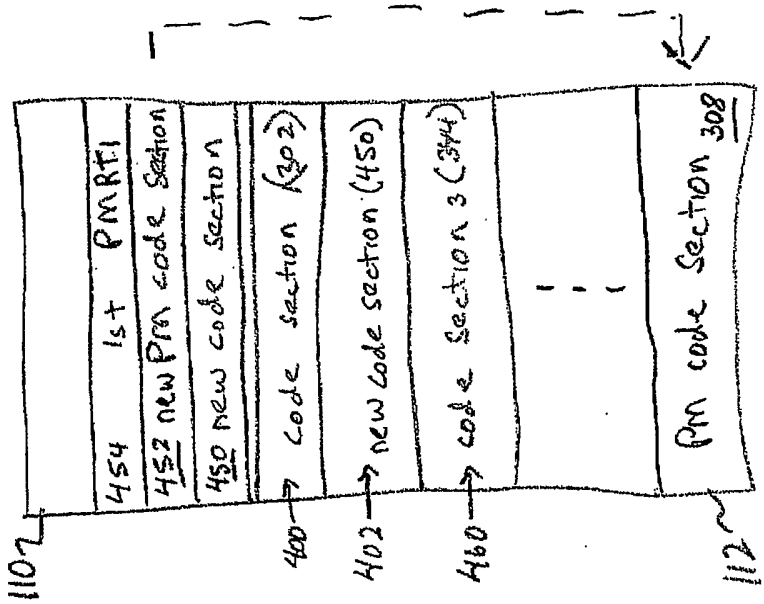
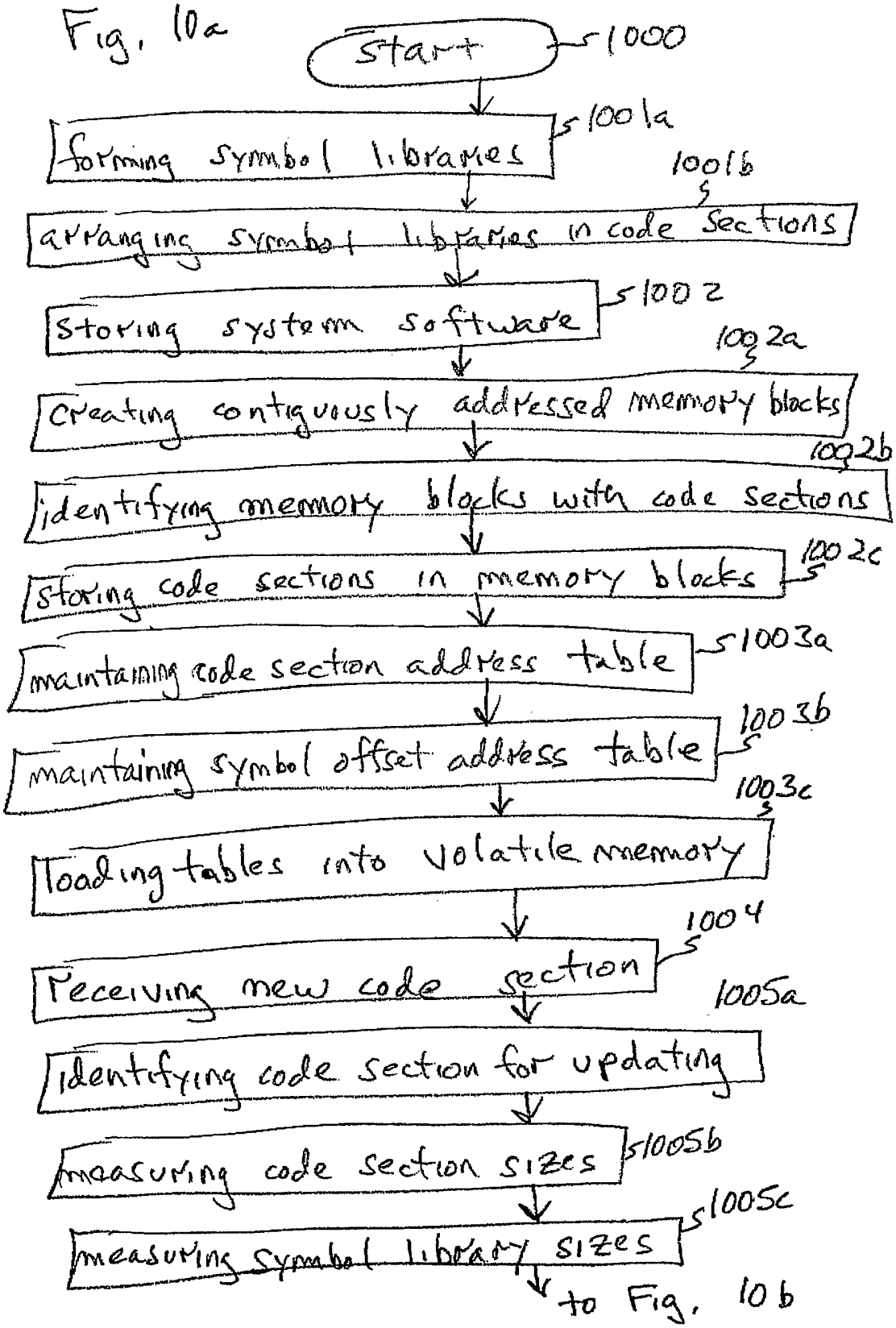

114
compactor

Fig. 9e


110
454 1st PMRTI
452 new PM code section
450 new code section
400 code section (302)
402 new code section (450)
460 code section 3 (344)
112 PM code section 308

14/16

Fig. 10a


start
1000
forming symbol libraries
1001a
arranging symbol libraries in code sections
1001b
storing system software
1002
creating contiguously addressed memory blocks
1002a
identifying memory blocks with code sections
1002b
storing code sections in memory blocks
1002c
maintaining code section address table
1003a
maintaining symbol offset address table
1003b
loading tables into volatile memory
1003c
receiving new code section
1004
identifying code section for updating
1005a
measuring code section sizes
1005b
measuring symbol library sizes
1005c
to Fig. 10b

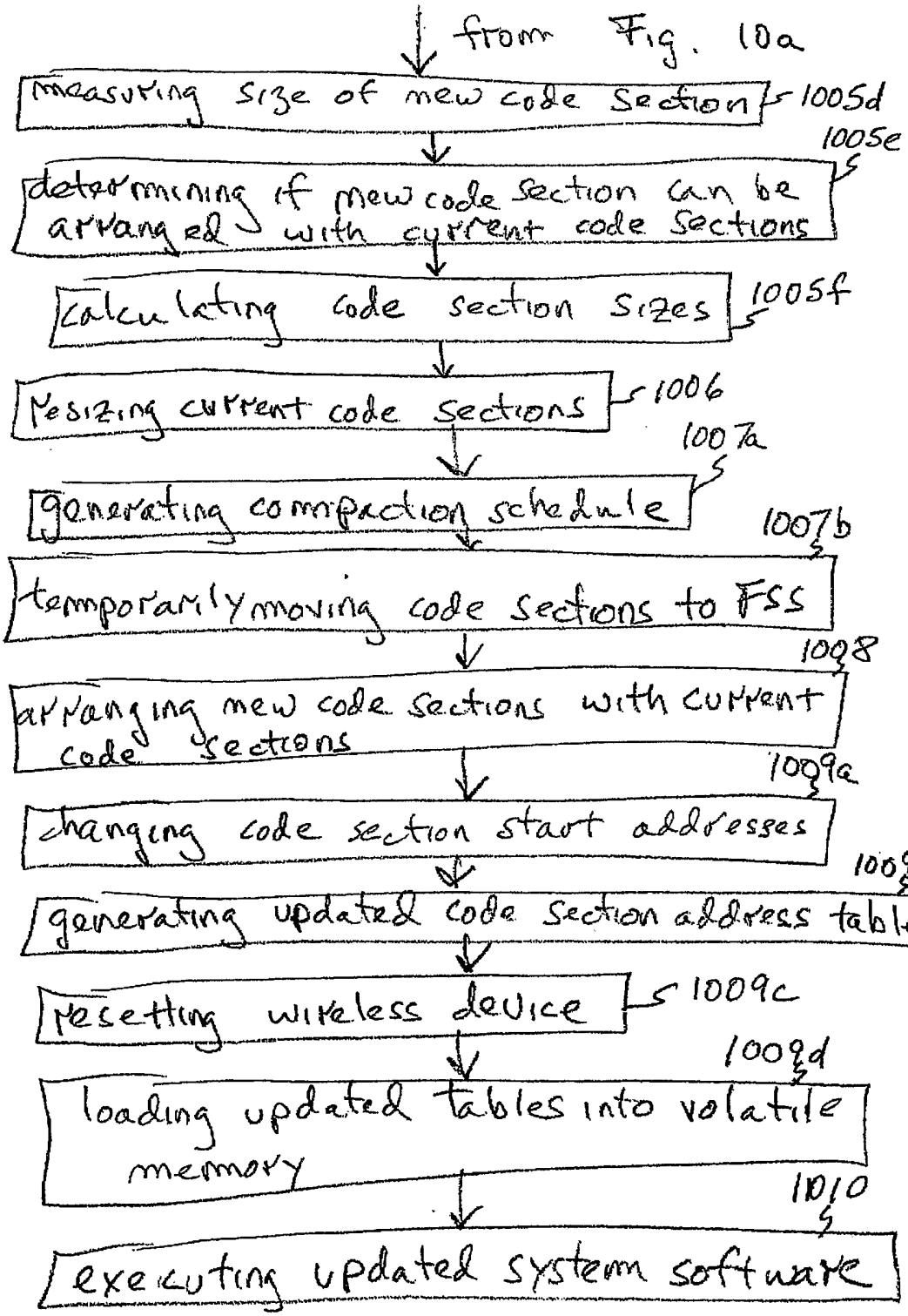
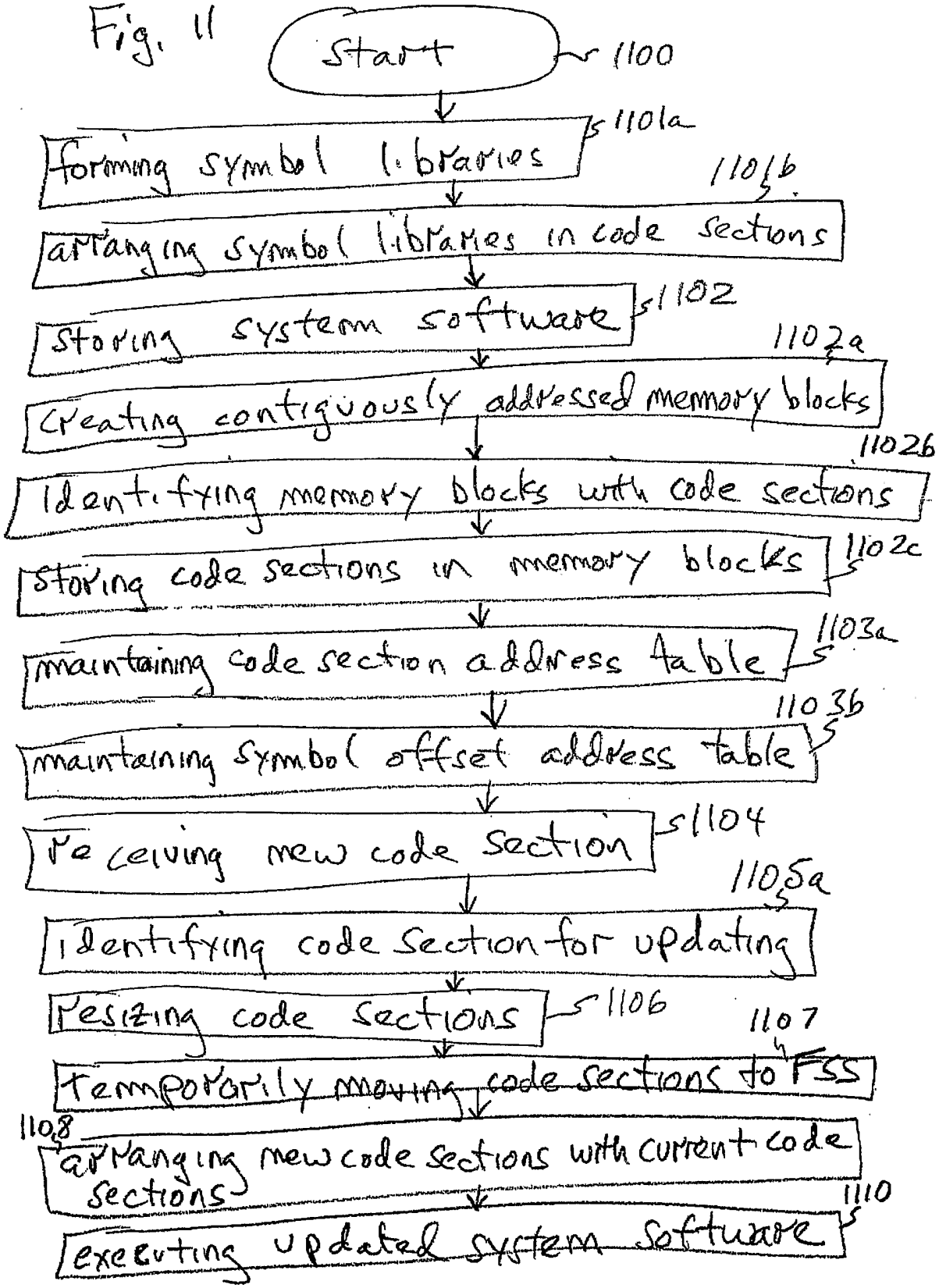

from Fig. 10a
measuring size of new code section
1005d
determining if new code section can be arranged with current code sections
1005e
calculating code section sizes
1005f
resizing current code sections
1006
generating compaction schedule
1007a
temporarily moving code sections to FSS
1007b
arranging new code sections with current code sections
1008
changing code section start addresses
1009a
generating updated code section address table
1009b
resetting wireless device
1009c
loading updated tables into volatile memory
1009d
executing updated system software
1010

Fig. 10b

Fig. 11


Start
1100
forming symbol libraries
1101a
arranging symbol libraries in code sections
1101b
storing system software
1102
creating contiguously addressed memory blocks
1102a
identifying memory blocks with code sections
1102b
storing code sections in memory blocks
1102c
maintaining code section address table
1103a
maintaining symbol offset address table
1103b
receiving new code section
1104
identifying code section for updating
1105a
resizing code sections
1106
temporarily moving code sections to FSS
1107
arranging new code sections with current code sections
1108
executing updated system software
1110