# EXHIBIT 31

RO888-0264 CGR G. D. Batalden, R. A. Christie, C. S. Graham, L. E. Gregg and F. P. Sheppard

IBM Technical Disclosure Bulletin Vol. 32 No. 9A February 1990

# 90A060497

## MAINTAINABLE ROS CODE THROUGH THE COMBINATION OF ROM AND EEPROM

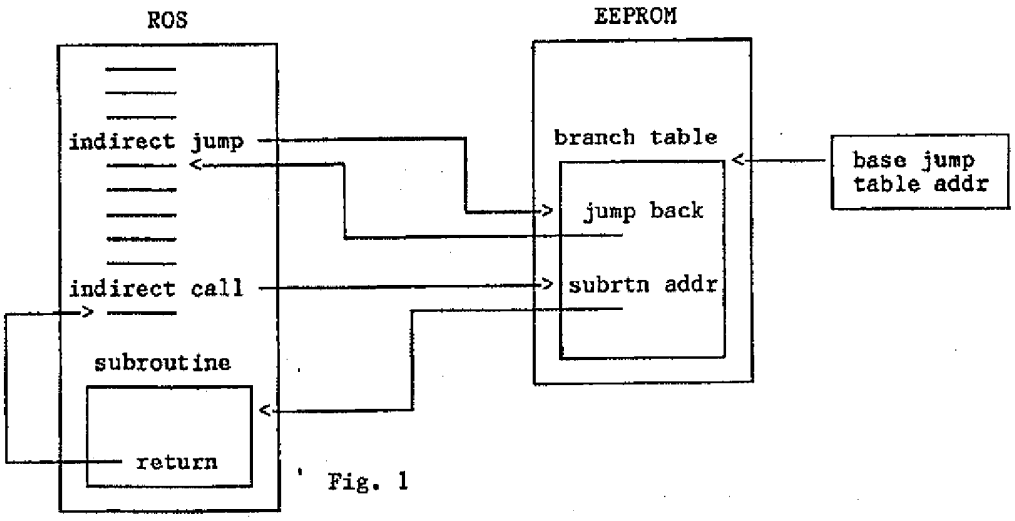


Fig. 1

A processor is disclosed where most of the diagnostic and IPL code continues to reside in ROS while patches and updates reside in EEPROM.

Code normally in ROS is first examined for logical break points where a new or changed function might be required. At each of these points, an indirect jump is inserted. (An indirect jump goes to where the jump-to location points. Example: Assume that location 100 contains 2304. An indirect jump-to "address 100" would go not to 100, but to the address contained in location 100, or 2304.) Also, the addresses of commonly used subroutines are placed in the table. Subroutines are then reached with an indirect call. This approach supports a high-level design concept while taking advantage of the normal method for passing control. Note then that all of these jumps and calls are now indirect. The "real" target addresses are listed in a table. The table is, in turn, identified by a pointer in a register. While the code continues to reside in ROS, the jump table and some free area used for code updates is placed in the EEPROM. (A copy of this original jump table is also placed in the ROS for backup--which will be explained in more detail later.) It is this technique which retains the advantages of ROS while adding the flexibility of EEPROM.

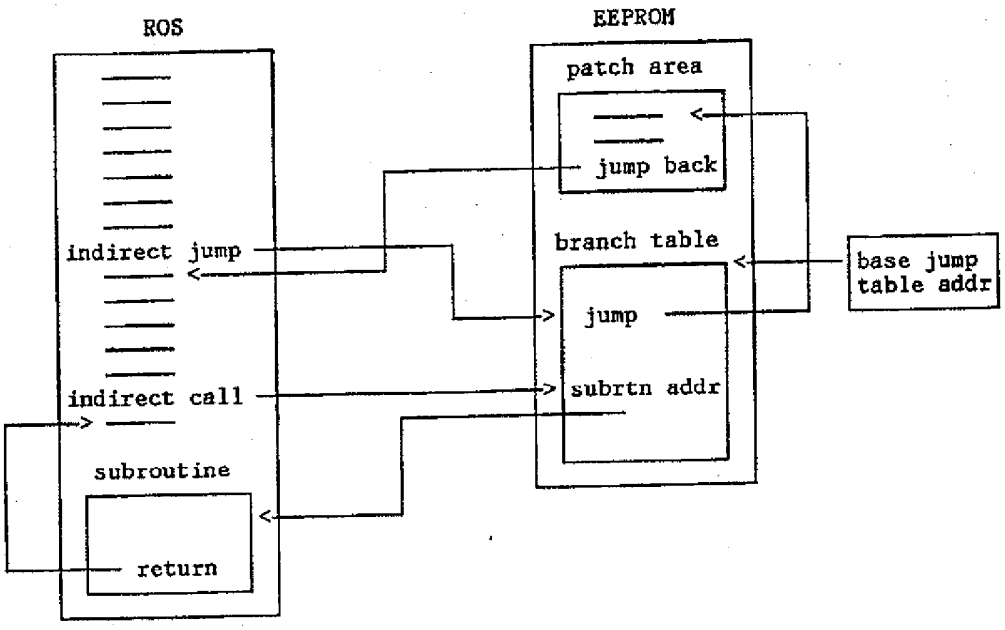


Fig. 2

The processor begins execution with diagnostic code in the ROS. When one of the indirect jumps is encountered, the EEPROM is used to find the target address. If no additional code has been needed, or no patches required, the jump returns back to the next instruction in ROS (see Fig. 1).

If, however, there is new or changed code, the address in the EEPROM points to that updated code. At the completion of the updated code, execution returns to the ROS. New or patched code is placed in the free area of the EEPROM. Because the EEPROM will retain its contents even without power, once code is placed there, it will remain--much like the ROS code. A picture of this logical flow with patches in EEPROM follows (see Fig. 2).

In a very similar way, the EEPROM can be used to patch or update data areas. Data values which might be subject to future change can be located via pointers exactly like code sequences are located via pointers. If, in the future, there is a need to update the data value, then the new value can be placed in the EEPROM and the pointer modified to address the updated value.

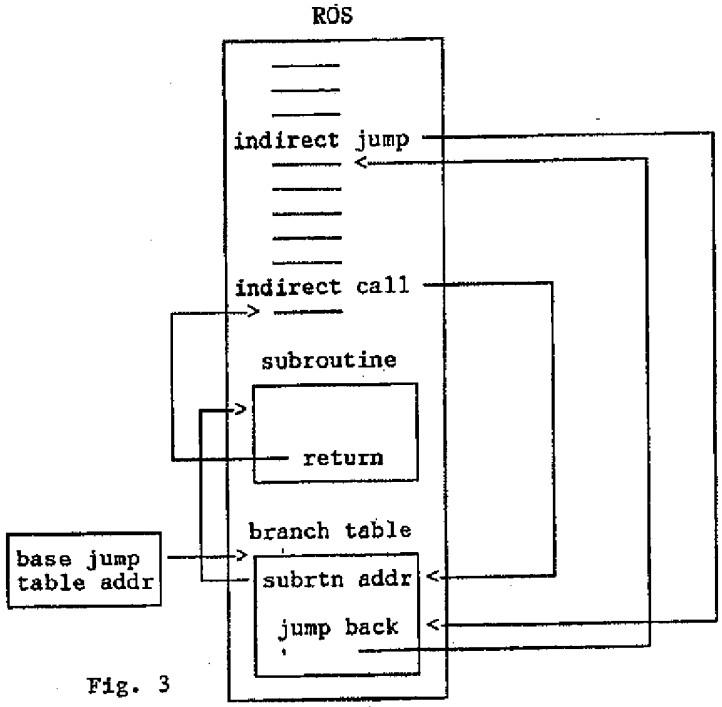


Fig. 3

The ability to update data can be used for system- or device-dependent parameters. Examples might include either some device-dependent characteristics of a disk drive or the number of drives which are attached.

A similar use of data modification would be maintaining a list of code modules which must be loaded to initialize the processor. This allows easy customization of the processor code load. If a new device were added later which required its own unique code in the processor, the module identification could be added to the list of items to load. Then, when the processor is subsequently loaded and prepared for operation, the code for the new device would "automatically" be loaded also.

The techniques described here also address the possibility of an invalid EEPROM. As mentioned several times, it is possible to write into the EEPROM. Therefore, it is possible to accidentally or even maliciously destroy the contents of the EEPROM. While unlikely, this cannot be ignored because the consequences would be a machine which cannot be loaded. This is handled first by a verification of the EEPROM during initial system checkout before it is used. If the EEPROM is suspect, it will not be used.

Verification of the EEPROM is done by adding together the contents of all words in the EEPROM to produce a checksum. If the checksum is correct, then the pointer to the branch table is set with the address of the table in the EEPROM. If, however, the EEPROM does not check out, the address of a backup branch table in the ROS is used. This table in ROS contains only addresses for the code originally placed in ROS. In other words, utilization of the ROS table will result in execution of only the originally supplied ROS code. This might include some errors, and it would obviously not include any enhancements made later. However, it would definitely be adequate to start the system and bring it up to a point where the updated EEPROM code could again be loaded. This backup table in ROS is an easy way to overcome the possible erasure of the EEPROM and guarantee that the system can be started. A picture of the logical flow using the branch table in ROS is shown in Fig. 3.