How did MS-DOS report error codes?

devblogs.microsoft.com/oldnewthing/20050117-00

January 17, 2005

Raymond Chen

The old MS-DOS function calls (ah, int 21h), typically indicated error by returning with carry set and putting the error code in the AX register. These error codes will look awfully familiar today: They are the same error codes that Windows uses. All the small-valued error codes like ERROR_FILE_NOT_FOUND go back to MS-DOS (and possibly even further back). Error code numbers are a major compatibility problem, because you cannot easily add new error code numbers without breaking existing programs. For example, it became well-known that "The only errors that can be returned from a failed call to OpenFile are 3 (path not found), 4 (too many open files), and 5 (access denied)." If MS-DOS ever returned an error code not on that list, programs would crash because they used the error number as an index into a function table without doing a range check first. Returning a new error like 32 (sharing violation) meant that the programs would jump to a random address and die. More about error number compatibility next time. When it became necessary to add new error codes, compatibility demanded that the error codes returned by the functions not change. Therefore, if a new type of error occurred (for example, a sharing violation), one of the previous "well-known" error codes was selected that had the most similar meaning and that was returned as the error code. (For "sharing violation", the best match is probably "access denied".) Programs which were "in the know" could call a new function called "get extended error" which returned one of the newfangled error codes (in this case, 32 for sharing violation). The "get extended error" function returned other pieces of information. It gave you an "error class" which gave you a vague idea of what type of problem it is (out of resources? physical media failure? system configuration error?), an "error locus" which told you what type of device caused the problem (floppy? serial? memory?), and what I found to be the most interesting meta-information, the "suggested action". Suggested actions were things like "pause, then retry" (for temporary conditions), "ask user to re-enter input" (for example, file not found), or even "ask user for remedial action" (for example, check that the disk is properly inserted). The purpose of these meta-error values is to allow a program to recover when faced with an error code it doesn't understand. You could at least follow the meta-data to have an idea of what type of error it was (error class), where the error occurred (error locus), and what you probably should do in response to it (suggested action). Sadly, this type of rich error information was lost when 16-bit programming was abandoned. Now you get an error code or an exception and you'd better know what to do with it. For example,

if you call some function and an error comes back, how do you know whether the error was a logic error in your program (using a handle after closing it, say) or was something that is externally-induced (for example, remote server timed out)? You don't.

This is particularly gruesome for exception-based programming. When you catch an exception, you can't tell by looking at it whether it's something that genuinely should crash the program (due to an internal logic error – a null reference exception, for example) or something that does not betray any error in your program but was caused externally (connection failed, file not found, sharing violation).

Raymond Chen

Follow