

The Wayback Machine - [https://web.archive.org/web/20000304044656/http://msdn.microsoft.com:80/library/backgrnd/html/msdn\\_c7pc...](https://web.archive.org/web/20000304044656/http://msdn.microsoft.com:80/library/backgrnd/html/msdn_c7pc...)

 **show toc**

## Microsoft P-Code Technology

Andy Padawer, Code Generator Manager  
Microsoft Development Tools Division

Created: April 1992

*Andy Padawer is Manager of Microsoft Development Tools Compiler Technology group, responsible for the design and development of Microsoft code generators. This includes the development of optimization, inlining, and p-code technology included in Microsoft C/C++ version 7.0. Previously at Microsoft, Padawer was the design and development lead for the QuickC® family of products, and developer of the QuickC code generator. He started at Microsoft in 1982 after graduating from Harvard University with an AB degree in Applied Math and Computer Science, and has been Code Generation Manager since 1990.*

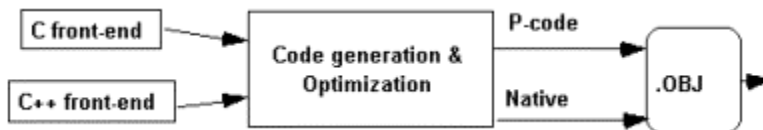
### Overview

As operating environments offer more sophisticated services and customers demand more features from applications, the size of executables continues to grow. This represents a problem for programmers who develop for both the Microsoft® Windows™ and MS-DOS® operating systems and who must respond to market pressures to limit the amount of memory consumed by applications.

Microsoft has introduced a code compression technology in its C/C++ Development System for Windows version 7.0 (C/C++ 7.0) called *p-code* (short for packed code) that provides programmers with a flexible and easy-to-implement solution for minimizing an application's memory requirements. In most cases, p-code can reduce the size of an executable file by about 40 percent. For example, the Windows Project Manager version 1.0 (resource files not included) shrinks from 932K (C/C++ 7.0 with size optimizations turned on) to 556K when p-code is employed.

Until now, p-code has been a proprietary technology developed by the applications group at Microsoft and used on a variety of internal projects. The retail releases of Microsoft Excel, Word, PowerPoint®, and other applications employ this technology to provide extensive breadth of functionality without consuming inordinate amounts of memory.

P-code works by compiling an application into an intermediate code format that is much more compact than 80x86 machine code. At link time, a small engine is built into your application that processes the p-code into native machine code during run time. Although there is an associated reduction in performance due to the extra step of interpretation, some simple techniques can minimize this effect.



Microsoft's p-code technology is implemented in the code-generation phase of the compiler. This means there are no syntactic or semantic considerations for the developer—that is, developers write their code in C or C++, and p-code works like any other "optimization" option.

### Key Features and Advantages

P-code offers the following benefits to programmers:

- **Easy to use.** Even though p-code offers program size reduction of about 40 percent in most cases, it involves almost no work on the part of the programmer to achieve this compaction. If desired, a compiler switch can be turned on to apply p-code throughout the entire program. P-code therefore provides a significant advantage over other program reduction techniques because the size savings are significant and the effort required is small. In addition, p-code can be used with the Microsoft Source Profiler and CodeView® debugger.
- **Supports full C/C++ 7.0 syntax.**
- **High degree of "tunability."** P-code can be applied selectively throughout a program through the use of pragmas. Size-critical functions and modules can be compiled with p-code, while speed-critical functions and modules can be compiled with the normal optimizations. Experimenting with these tradeoffs allows programmers to achieve the right size and speed balance.
- **Supported in programs for both Windows and MS-DOS.** Programs written for either the MS-DOS or Windows environment can be compiled with p-code. It therefore represents a code minimization solution for the broadest range of programs.

This paper describes p-code technology in detail, including how it works, some specifics of its implementation, and strategies for its use.

### Using P-Code

A significant advantage to p-code technology is the small amount of work required from the programmer. The fastest usage of p-code involves turning on a compiler switch and recompiling your application. P-code is then applied globally throughout the application. Alternately, the programmer can locally apply pragmas in specific modules or functions to turn p-code on and off where needed. This section describes both techniques.

### Global Use of P-Code

Global use of p-code involves the invocation of a compiler switch that converts every line of an application to p-code. The assumption made

by the compiler is that size reductions are of paramount importance and should never be sacrificed for a gain in execution speed.

Applications that experience a high percentage of CPU idle time are good candidates for global use of p-code. These include programs that are dominated by user-interface functions, such as word processors, electronic calendars, and small-business accounting packages. In these cases, the difference in speed to the user can be negligible. Other candidates for global use of p-code are programs that operate in batch mode where time of completion is not important.

For many applications, this "all-or-nothing" technique results in less than satisfactory performance because speed-critical functions are being compromised. For these situations, p-code should be applied selectively, as discussed in the next section.

## Local Use of P-Code

Local use of p-code involves the selective placement of pragmas in the application's source code, indicating to the compiler which sections are to be compiled as p-code and which are to be compiled as native machine code.

Pragmas can be placed either at the module or function level. For example:

```
// An example of p-code pragmas
#pragma optimize("q", on)      //Compile the following function using p-code:
Func1()
{
    // Code that can trade size for speed
}
#pragma optimize("q", off)      //Compile the following function with p-code
                                //turned off
Func2()
{
    // Speed-critical code
}
```

The following general guidelines apply when using p-code pragmas:

- Speed-critical functions should be compiled as native code. Also, routines that are called frequently (such as those appearing mostly within loops) should be compiled as native code even though the function code itself may not be CPU-intensive.
- User-interface routines such as menu and dialog handlers can be compiled as p-code. The perceived difference in their execution speed is usually negligible.
- Infrequently used routines should be compiled as p-code. These include routines such as error handling procedures and features of the application's functionality that are not used on a regular basis.

As with any other optimization, a profiler is the best tool for determining actual execution times. The most effective strategy is to do a complete function-level profile to provide the most accurate "map" of a program's execution speed. P-code can then be applied where appropriate.

## Performance Analysis

The nature of the p-code model is such that a distinct size advantage is gained at the expense of some execution speed. However, the extent to which speed is reduced depends on system factors as well as how the programmer uses p-code technology.

### System Factors

Although the p-code engine is inherently slower than direct CPU execution of machine code, various system-oriented factors can offset at least some of the speed reduction. This is because the manner in which a computer executes a particular program depends on the size of the program.

As program size increases, there is a greater tendency to trigger system mechanisms that can introduce execution delays. Two examples are virtual memory and caching programs. Large applications tend to induce more virtual page swapping as memory resources run low, and the relative frequency of cache hits will tend to decrease. These factors can dramatically reduce a program's speed.

Although p-code programs will execute more slowly at the instruction level, overall system throughput tends to fare much better. For very large programs in memory-constrained environments, the execution speed difference between p-code and non-p-code versions of an application are less significant.

In simple terms, any loss of speed in the final code is often offset by the significant reductions in code size, especially on systems such as Windows, where reduced code size reduces the amount of swapping.

## P-Code Internals

### How the Engine Works

If you use p-code in any part of your C/C++ 7.0 application, the linker automatically binds in a simple copy of the p-code run-time engine. This engine adds approximately 9K to the size of your executable file.

The p-code engine is a relatively simple machine that processes a series of "high-level" operation codes ("opcodes"). It is *stack-based*, meaning that it uses the system stack for virtually all its operations. In contrast, an actual microprocessor uses registers for most of its operations and uses the stack primarily to perform function call mechanics. All operands used by p-code instructions are stored on the stack.

P-code instructions are much more compact than assembly instructions because it is not always necessary to specify the source and/or destination addresses for each instruction. For example, to add two register-resident values using assembly language, you would use the following syntax:

```
ex = add ax, bx
```

The source registers AX and BX are added and placed in the destination EX. The same instruction in p-code is simply:

```
ex = AddW
```

Since each p-code instruction implicitly pops its operands off the stack and pushes its result back onto the stack, the single AddW opcode encapsulates both the locations of the operands (on the stack) as well as the stack mechanics. Fewer instructions are needed to represent the opcode because much of the process is performed by default. The AddW instruction above is actually equivalent to the following assembly language sequence:

```
pop cx      ; Pop first operand from the stack into cx
pop di      ; Pop second operand from stack into di
add di,cx   ; Add the two values, store result in di (by default)
push di     ; Push the result back onto the stack
```

In cases where a p-code instruction must modify the value of a variable, source and destination addresses are specified in the opcode because variables cannot be stored on the stack. However, most instructions use the stack for at least one of their arguments.

In addition to implicit stack mechanics, further size reductions are gained through the use of *assumed values*. Since many operations involve the same specific values over and over, several opcodes have been created to incorporate these values without using operands. For example, consider a statement using the Jump On Not Equal instruction:

```
EX = JneWb 05
```

This statement pops two words off the stack and compares them. If they are not equal, a jump of length 5 is performed. The "b" in the instruction indicates that a 1-byte operand is required. The following alternate form of this instruction assumes that a jump of length 5 is required, thus eliminating the additional space needed for the operand:

```
EX = JneW5
```

## Opcode Format and Statistics

The p-code engine's use of implied addressing enables an opcode size that averages less than 2 bytes. Because of this, two sets of opcodes are defined: standard and extended.

The standard set consists of the 255 most commonly used opcodes. These opcodes are a single byte in length and can be used in combination of up to 4 bytes of data. The extended set consists of 256 opcodes that are used less frequently. The following table shows run-time statistics for the p-code opcode sizes in a sample program of 200,000 lines of C source (.c files, does not include .h files) compiled into all p-code. Note that this program consisted almost exclusively of 1- and 2-byte opcodes, with 3- and 4-byte opcodes representing a very small percentage.

P-code Opcode Size (bytes)	Number of Times Used	Frequency of Usage (%)	% of Code Size
1	414,369	55.5	37.0
2	321,908	39.2	52.0
3	41,838	4.6	9.1
4	8,782	.71	1.8

## Further Optimization Through Quoting

An important feature of p-code technology is *quoting*. Quoting enables the sharing of a single instance of a code sequence. Quoting is similar to using routines in a high-level language because it allows a single block of code to be used throughout the program without incurring added space. In order to implement this feature, the compiler examines the code that it generates, looking for places where a sequence of instructions is repeated. If it finds such repetitions, it replaces all but one of the occurrences with a jump instruction that directs the flow of execution to the beginning of the quoted block of code. Quoting provides approximately 5 to 10 percent additional compression in an executable. As with p-code in general, quoting can be controlled at either the global or program level. Otherwise there is no additional programmer involvement needed. Since quoting involves many jumps to labels, it can make compiled code difficult to read and debug. A good strategy is to turn quoting off during program development, and then turn it on once the program has been fully debugged.

Quoting differs from function calls in that there are no arguments and no return value. Only the path of execution is changed. To implement quoting, two instructions are used: QUOTE and EQUOTE. The QUOTE instruction takes a 1- or 2-byte offset as an argument. When a QUOTE is executed, it saves the address of the next instruction as a return address and performs a jump to the specified offset. Instead of pushing the return address onto the stack, it is stored in the PQ register. When an EQUOTE instruction is executed, it checks whether PQ contains an address. If not, EQUOTE does nothing; if it does, a jump is performed back to that address. This allows the quoted section of code to be executed in one of two ways: in sequence with the preceding and following code, or as a quote call.

Function calls within code blocks can be quoted, and even interprocedural quoting is supported. Nested quotes, however, are not supported.

In the following example, two lines of source code containing a common subexpression **i+j+func()** appear within the same module. During

the first instance of the code, the quote is marked with a starting label of L1 and a standard end label of EQuote. When the code is used again in the second instance, p-code generates a call to the quote, rather than producing the entire sequence all over again:

Source code	P-code
m = i+j+func();	L1: LdfW i
	LdfW j
	AddW
	CallFCW func
	AddW
	EQuote
n = i + j + func();	StfW m
	Quote L1
	StfW n

This technique is similar to the frequently used *common subexpression reduction* method employed by many optimizing compilers. In this case, the common subexpression **i+j+func()** is converted to a reusable "routine."

### Native Entry Points

If pragmas are used to turn p-code on and off throughout a program, it is possible that a machine code function will call a p-code function. When this happens, the program must stop executing machine code and turn control over to the p-code engine, which can then execute the p-code.

For this to occur, a p-code function normally contains a *native entry point* at its beginning. The native entry point consists of several machine code instructions that transfer control to the p-code engine. For each function that is specified as p-code, the compiler automatically generates the necessary entry sequence. There is a small overhead of about 6 bytes associated with the entry sequence. You can instruct the compiler to suppress generation of these entry sequences if you are sure that no machine code function will call a p-code function. This suppression can be done globally through the **Gn** switch or locally by placing the **#pragma native\_caller(off)** statement before the p-code function for which you want to remove the call sequence.

### Debugging P-Code

Applications compiled with p-code can be debugged using the Microsoft CodeView debugger in virtually the same manner as an application compiled without p-code. Both source-level and assembly-level debugging of p-code applications are supported. In assembly mode, p-code undergoes a special disassembly process that shows p-code instructions rather than native assembly instructions.

When a p-code program halts at a breakpoint, the register window changes to show the stack and engine state. All normal CodeView debugger commands, such as break, step, watch, and others, work identically for both p-code and non-p-code applications.

The Options/Native menu item in the CodeView debugger allows you to disable p-code support, and to work only with machine code. If you choose to debug with p-code on, you can even single-step through the p-code engine itself.

### Summary

P-code technology provides programmers who develop for both MS-DOS and Windows operating systems a new way to shrink the size of their executables by an average of 40 percent. Although there is a performance trade-off, this can be minimized through the effective use of p-code on "idle-time" routines such as those that handle the user interface. Microsoft uses this technique in many of its own applications.

A major strength of this code compression technology is its high payoff and low investment of time on the part of the programmer. P-code can be implemented globally throughout an application simply by recompiling. Placement of pragmas before strategic routines ensures that code compression is maximized while performance loss is minimized.

---

[Send feedback](#) on this article. Find [support options](#).

© 2000 Microsoft Corporation. All rights reserved. [Terms of use](#).