OPTION PROFILING and OPTION TRACECACHE

Overview

PicoMite MMBasic provides two complementary performance tools: **OPTION PROFILING**, which counts how often each statement executes, and **OPTION TRACECACHE**, which caches the compiled form of frequently-executed statements so they can be replayed without re-parsing the BASIC source.

Used together they let you identify hot loops and then accelerate them with zero changes to your BASIC program.

Build Availability

The two features have different build coverage.

OPTION PROFILING is available in every build variant.

OPTION TRACECACHE (and the DO-loop fast condition path) is only compiled into builds that define the `CACHE` flag. These are:

Build	MCU	OPTION PROFILING	OPTION TRACECACHE
PICO	RP2040	Yes	Yes
PICOUSB	RP2040	Yes	—
PICOMIN	RP2040	Yes	—
VGA	RP2040	Yes	—
VGAUSB	RP2040	Yes	—
WEB	RP2040	Yes	—
PICORP2350	RP2350	Yes	Yes
PICOUSBRP2350	RP2350	Yes	Yes
VGARP2350	RP2350	Yes	Yes
VGAUSBRP2350	RP2350	Yes	Yes
WEBRP2350	RP2350	Yes	Yes
HDMI	RP2350	Yes	Yes
HDMIUSB	RP2350	Yes	Yes

On builds without CACHE, `OPTION TRACECACHE` and `OPTION CACHE` commands are not compiled in and will produce a syntax error if used. All other sections of this document apply only to CACHE-enabled builds.

OPTION PROFILING

Syntax

```
OPTION PROFILING ON
OPTION PROFILING OFF
```

Description

`OPTION PROFILING ON` allocates per-command and per-sub/function execution counters. The counters are incremented every time a statement executes. `OPTION PROFILING OFF` frees the counter memory and stops counting.

When the program ends (the `END` statement executes), a `[PERF]` report is printed listing the 20 most-executed commands and the 20 subs/functions with the most total statement executions. If OPTION TRACECACHE is also active, cache hit/miss statistics are included in the same report.

Profiling is zero-overhead when disabled: no code runs and no memory is allocated.

Example

```
Option Profiling On
' ... run your program ...
End
' [PERF] report printed at END
```

OPTION TRACECACHE

Syntax

```
OPTION TRACECACHE ON [size [, flags]]
OPTION TRACECACHE OFF
OPTION CACHE DEBUG ON
OPTION CACHE DEBUG OFF
OPTION CACHE SUB name [, name ...]
OPTION CACHE SUB OFF
```

Description

The trace cache compiles the first execution of each statement into a compact bytecode, then replays that bytecode on every subsequent hit. Replaying eliminates variable-name hash lookups, operator-table indirections, and recursive descent parsing for the statement.

The cache uses an open-addressed hash table keyed by the source-token pointer (the address of the statement in program memory). Entries are stable for the life of the loaded program; any event that restructures program memory or the variable table (`NEW`, `DIM`, `ERASE`, `OPTION BASE`, `OPTION EXPLICIT`) invalidates all entries.

Memory is allocated lazily on the first cached execution. `OPTION TRACECACHE OFF` releases the memory immediately. Typical size with default settings is ~13.5 KB total (1.5 KB table + 12 KB arena).

Parameters

Parameter	Description
size	Hash table slot count. Must be a power of two. Clamped to 16-4096. Default: 64.
flags	Feature bitmask (TCF_* bits, see table below). Default: 0x3F (all features).

Feature Flags

Value	Name	Description
0x01	TCF_LET_NUM	Numeric scalar/array LET assignments and INC statements
0x02	TCF_LET_STR	String scalar LET assignments
0x04	TCF_IF	IF condition caching and SELECT CASE structure cache
0x08	TCF_LOOP	DO WHILE/UNTIL condition caching (TraceCacheTryIf path)
0x10	TCF_JUMP	GOTO/GOSUB target caching and CASE/CASE ELSE body-exit jumps
0x20	TCF_RESTORE	RESTORE target caching
0x3F	TCF_ALL	All features (default)

To enable only numeric LET and IF caching: `OPTION TRACECACHE ON 64, 5`

Supported Statement Forms

Numeric LET (TCF_LET_NUM)

The left-hand side may be a plain numeric scalar or a 1-D or 2-D array element. Variables may be global or `LOCAL`, of type integer or float. `T_CONST` variables and struct members

are not supported.

```
lhs = const
lhs = rhs_var
lhs = operand BINOP operand
lhs = f(operand)
lhs = array(i)
lhs(i) = expr
lhs(i, j) = expr
```

Supported Binary Operators

Type	Operators
Float (NBR)	+ - * / ^
Integer (INT)	+ - * \ MOD AND OR XOR << >>
Comparisons	= <> < > <= >= (NBR and INT)

Supported 1-Argument Intrinsics

`SIN` `COS` `TAN` `ASIN` `ACOS` `ATAN` `SQR` `ABS` `INT` `EXP` `LOG` `SGN` `FIX` `CINT` `DEG` `RAD`

Trig intrinsics bail to the interpreter when `OPTION ANGLE DEGREES` is in effect. `LOG` bails when the argument is <= 0.

Supported 2-Argument Intrinsics

`ATAN2(y, x)` `MAX(a, b)` `MIN(a, b)`

Supported 0-Argument Functions

`PI` — folded to a compile-time constant; no runtime call.

`Rnd` — evaluated at replay time, advancing the RNG state identically to the normal interpreter. Both `Rnd` (no-bracket form) in arbitrary expressions and as a sub-expression of INT, MAX, etc. are supported.

Example expressions that cache

```
x = a * b + c
y(i) = SIN(angle) * radius
z = Rnd * 100
styy(st) = (Rnd * h >> 1) - (h >> 2)
stz(st) = Int(Rnd * 5 + 5) / 10 + f
result = MAX(a, b) + MIN(c, d)
```

INC Statement (TCF_LET_NUM)

`INC` is cached under the same `TCF_LET_NUM` flag as numeric LET. Both constant and variable steps are supported.

```
Inc var           ' step = 1 (constant)
Inc var, const    ' step = numeric literal
Inc var, stepvar  ' step = plain scalar variable
```

The target and step may be global or local integer or float scalars. Array elements, struct members, string variables, and expression steps are not supported and fall back to the normal interpreter.

When a variable step is used (`Inc i, j`), both `i` and `j` are resolved to direct memory pointers at compile time. On every replay only two pointer dereferences are needed — no hash lookup for either variable.

String LET (TCF_LET_STR)

The left-hand side must be a plain string scalar (not an array element or by-reference parameter).

```
s$ = "literal"        ' simple literal assign (OptionEscape must be off)
s$ = t$               ' string copy
s$ = s$ + "literal"   ' in-place append literal
s$ = s$ + t$          ' in-place append variable
```

The left operand of `+` must be the same variable as the LHS. If the result would overflow `STRINGSIZE` the cache bails to the interpreter (which raises the same error).

IF Condition Caching (TCF_IF)

Single-line IF conditions (everything up to `THEN` or `GOTO`) are compiled and cached. The compiled condition is replayed as a boolean (true/false) result.

```
If x > 0 Then ...
If a = b And c < d Then ...
```

DO Loop Fast Condition Path

In addition to the general IF caching, the DO loop engine pre-compiles a specialised "fast condition" at `DO` setup time when the loop condition is a simple `var OP const` or `const OP var` comparison. This fast path is stored directly in the DO stack entry (not in the trace cache hash table) and is evaluated in `cmd_loop` with a single pointer dereference and integer/float compare — no hash probe, no replay machinery.

The fast condition path is activated automatically whenever `#ifdef CACHE` is defined, independent of the `TCF_LOOP` flag. It supports all six comparison operators (`<` `>` `<=` `>=`

`=` `<>`) for both integer and float variables, and correctly re-resolves local variables when the call frame changes.

```
Do While i < 100000      ' fast path: i directly compared to 100000
  Inc i
Loop

Do Until x >= 4.0        ' fast path: x directly compared to 4.0
  x = x + 0.1
Loop
```

For more complex conditions (e.g., compound expressions with `And` / `Or`) the `TCF_LOOP` general IF cache is used instead.

SELECT CASE Structure Cache (TCF_IF)

On the first execution of a `SELECT CASE` statement, the cache scans the entire SELECT body once and builds a compact table of pre-compiled arm entries in the arena. On every subsequent hit, the selector value is compared directly against the pre-built table — no `GetNextCommand` scan, no `evaluate()` call per arm.

Supported arm forms (CASE values must be numeric literals):

```
Case value            ' exact match
Case lo To hi         ' range
Case < value          ' comparison (IS keyword optional)
Case Is >= value      ' comparison with IS
Case Else             ' default
```

Multiple comma-separated elements on one CASE line (e.g. `CASE 1, 3, 5`) are each stored as separate arms pointing to the same body. Arms are checked in source order, so the sequential semantics of overlapping IS conditions are preserved:

```
Select Case stz(st)
  Case < 0.2  : cl(st) = &HFFFF00   ' stz < 0.2
  Case < 0.6  : cl(st) = &HFFFFFF   ' 0.2 <= stz < 0.6
  Case < 0.8  : cl(st) = &HFFFF     ' 0.6 <= stz < 0.8
  Case Else   : cl(st) = &HFF       ' stz >= 0.8
End Select
```

String selectors and any CASE value that is not a plain numeric literal (e.g. a variable or function call) cause the entire SELECT to fall back to the normal interpreter permanently.

In addition, the jump from a completed CASE body back past `END SELECT` is cached via the existing jump cache (TCF_JUMP). On the second and subsequent executions, `CASE` / `CASE ELSE` body-exit jumps cost a single hash probe with no program scan.

GOTO / GOSUB / RESTORE Jump Caching (TCF_JUMP / TCF_RESTORE)

The resolved target address of `GOTO`, `GOSUB`, and `RESTORE` statements is cached after the first `findlabel` / `findline` call. Subsequent executions skip the linear scan entirely. Variable-argument `RESTORE` (where the target is a run-time expression) is not cached.

Auxiliary Commands

OPTION CACHE DEBUG ON | OFF

When debug mode is on, every statement that fails to compile prints a diagnostic line to the console:

```
[TC-BAD] (SubName): first ~60 chars of statement
```

Use this to find out why coverage is lower than expected. Common causes: unsupported operators, array/struct LHS, expressions with more than 8 operands, or string variables on the numeric path.

OPTION CACHE SUB name [, name ...] | OFF

Restricts the cache to statements lexically inside the listed sub/functions. Top-level code and unlisted subs fall through to the interpreter as normal.

`OPTION CACHE SUB OFF` clears the opt-in list and allows caching everywhere (the default).

```
Option Cache Sub MyHotLoop, AnotherSub
' Only statements inside MyHotLoop and AnotherSub are now cached
```

Performance Report (OPTION PROFILING ON + END)

When both OPTION PROFILING and OPTION TRACECACHE are active, the `[PERF]` report printed at `END` includes:

```
[PERF] tracecache: flags=0x3f size=64 replays=4200000 compiles_ok=12 compiles_b
[PERF] tracecache: lookup_null=0 alloc_fail=0 optin_skip=0 jump_hits=38
[PERF] tracecache hits by SUB (top 20):
        let_hits   if_hits     total  name
         1000000    500000   1500000  MyHotLoop
```

Counter	Meaning
replays	Total cache hits (LET + IF + INC + SELECT CASE)
compiles_ok	Statements successfully compiled
compiles_bad	Statements rejected (fell back forever)
lookup_null	Hash table full; slot could not be created
alloc_fail	Arena exhausted; payload could not be allocated
optin_skip	Statements skipped because sub not in opt-in list
jump_hits	GOTO/GOSUB/RESTORE cache hits

Typical Performance Gains

The table below shows measured loop times on an RP2040 at 252 MHz for a tight `DO UNTIL` loop running 1,000,000 iterations, compared to a `FOR` loop doing the same work.

Configuration	DO UNTIL	FOR
No cache	3822 ms	771 ms
DO fast condition only	2839 ms	774 ms
+ INC constant step cached	2204 ms	779 ms
+ INC variable step cached	~2200 ms	~780 ms

The DO fast condition path is always on (requires `CACHE` build flag). The INC optimisation requires `OPTION TRACECACHE ON` with `TCF_LET_NUM` (0x01) set.

Quick-Start Example

```
' At the top of your program:
Option Profiling On
Option Tracecache On   ' 64 slots, all features

' ... your program ...

End   ' prints [PERF] report showing hits by sub and cache statistics
```

To restrict caching to one hot sub while debugging:

```
Option Tracecache On 128
Option Cache Sub MyInnerLoop
```

```
Option Cache Debug On   ' show any TC-BAD lines
```

Memory Footprint

Setting	Approximate memory used
OPTION TRACECACHE ON (default, 64 slots)	~13.5 KB
OPTION TRACECACHE ON 128	~27 KB
OPTION TRACECACHE ON 256	~54 KB
OPTION PROFILING ON	8 bytes x MAXCMD + 8 bytes x MAXSUBFUN
OPTION TRACECACHE OFF	0 (freed immediately)

Arena size scales with slot count (ratio ~12x slot header size). Both the table and arena are freed by `OPTION TRACECACHE OFF` or by a program `NEW`.