OPTION PROFILING and OPTION TRACECACHE

PicoMite MMBasic - Performance Optimisation Guide

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	--
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
```

OPTION PROFILING and OPTION TRACECACHE

PicoMite MMBasic - Performance Optimisation Guide

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

OPTION PROFILING and OPTION TRACECACHE

PicoMite MMBasic - Performance Optimisation Guide

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_bad=2
[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.

Other Optimisations

The trace cache is the largest source of speedup but it sits on top of several other firmware optimisations that affect every program -- cached or not. The most useful ones to know about are listed here.

OPTION LOCAL VARIABLES n

```
OPTION LOCAL VARIABLES n       ' n = 32 .. MAXVARS-32
```

Splits the variable name hash table between local and global slots. The variable lookup that runs on every uncached statement (and on the first execution of every cached one) probes the local hash first, then the global hash; reducing the number of hash collisions in either domain speeds up every name lookup.

The default split (MAXLOCALVARS) is balanced for a typical program with a few dozen LOCALs per sub and a few hundred globals. Programs with very many globals and few LOCALs benefit from a smaller n; programs with deep sub trees and many LOCALs benefit from a larger n.

The setting must be issued before any DIM or LOCAL statement runs (typically at the very top of the program, before any sub call).

LIST COLLISIONS

```
List Collisions
```

Diagnostic that reports any variable name groups that hash to the same bucket in the current LOCAL or GLOBAL domain. Use this after a representative run to decide whether OPTION LOCAL VARIABLES n should be retuned, or whether a particular variable should be renamed to break a collision.

A program with no reported collisions is already at the lookup-speed floor for that hash configuration; further

speedups require the trace cache.

DO Loop Fast Condition Path

Always active on CACHE-enabled builds (no opt-in required). Documented above under OPTION TRACECACHE because it shares the trace-cache compiler, but the fast path itself runs straight out of the DO stack entry -- it does not consume a trace-cache slot or arena byte. Programs that disable OPTION TRACECACHE still benefit from it.

Jump Target Caching for CASE Body Exit

The jump from the end of a CASE body back past END SELECT is cached via TCF_JUMP. This is automatic whenever both TCF_IF and TCF_JUMP are set in the active flags (the default). It removes a GetNextCommand scan per CASE arm taken.

Auto-Resolution of First-Assignment Scalars

When OPTION EXPLICIT is off, the trace cache compiler creates an undeclared LHS scalar (e.g. count = 0 at top level on the first hit) instead of bailing for a retry. This lets simple counter / accumulator patterns reach ST_COMPILED on the first execution rather than the second. No change in BASIC semantics; the variable would be created on the same line by the interpreter anyway.

Local-Variable Re-Resolution by Frame Generation

Cached entries that reference LOCAL variables store the variable name, not its slot index. A 16-bit g_local_frame_gen counter is bumped on every ClearVars() (sub return, NEW, RUN, ERASE, etc.). On the first replay inside a new call frame the entry walks its op list once and re-resolves each LOCAL name to the new slot; subsequent replays in the same frame execute with no hash lookups. This is what makes the cache safe to use inside recursive subs and deep call trees.

Profiling Overhead

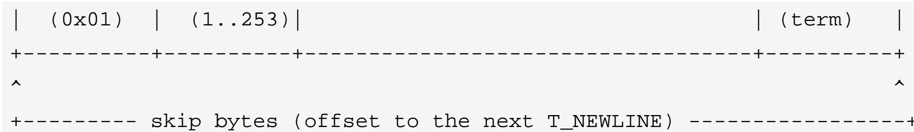
OPTION PROFILING ON adds a single 64-bit increment per executed statement and a single 64-bit increment per sub/function entry. There are no heap allocations after the initial counter arrays. Profiling can be left on in production if the report at END is wanted; the runtime cost is typically below 2 % on integer-heavy code.

When both OPTION PROFILING and OPTION TRACECACHE are active, the trace cache replay path is counted as a single statement execution. Cached and uncached statements appear identically in the per-command totals; the cache-hit columns in the [PERF] report show how many of those executions were served from the cache.

Tokenised Line Format with Skip Byte

Every program line in ProgMemory (and in the optional library area) starts with a 2-byte header: the T_NEWLINE marker (0x01) followed by a skip byte that records the in-flash length of the line -- i.e. the offset in bytes from this T_NEWLINE to the next T_NEWLINE or the end-of-program terminator.

```
+----------+----------+----------------------------------+----------+
| T_NEWLINE|  skip    |  line content (tokens, args, ...) |   0x00   |
```



```
|  (0x01)  |  (1..253)|                                  | (term)   |
+----------+----------+----------------------------------+----------+
^                                                                   ^
+--------- skip bytes (offset to the next T_NEWLINE) -----------------+
```

Encoding of the skip byte:

Value	Meaning
3 .. 0xFD (3..253)	Exact byte offset from T_NEWLINE to the next line header
0xFE (T_NEWLINE_SKIP_NONE)	Line too long for one byte (>253) -- fall back to structural scan
0 and 0xFF	Reserved (would conflict with the 0,0 end-of-program terminator and with flash-erased state)

The skip byte is written by the tokeniser (tokenise() in MMBasic.c) immediately after each line is compacted into tknbuf, by measuring from tknbuf[0] to the first 0,0 pair. Because the in-RAM/in-flash writers stop copying at the first 0,0, the value stored is the same length the writer will commit. The macro T_NEWLINE_HDR (= 2) is the size of the header and is what every p += T_NEWLINE_HDR; site advances.

A diagnostic verifier (g_verify_line_skip, Commands.c) walks the program and checks that every skip byte either accurately points at the next T_NEWLINE (or end-of-program) or is T_NEWLINE_SKIP_NONE. It is off by default and used during firmware development to validate the tokeniser.

How the skip byte is used to optimise processing

The skip byte turns several O(line length) byte-scans into O(1) jumps. In each case the slow path is preserved as a fallback when the skip byte is T_NEWLINE_SKIP_NONE (line too long) or otherwise unusable.

Comment-line fast path in ExecuteProgram (MMBasic.c). When the interpreter encounters a line whose first non-whitespace token is ' (apostrophe / REM-style comment), it normally has to walk the line byte by byte looking for the next T_NEWLINE. The fast path instead does:

```
if (line_skip valid && line_start[line_skip] == T_NEWLINE)
    p = line_start + line_skip;     /* O(1) jump to next line */
```

Programs with many full-line comments -- including documentation blocks, commented-out code, and the comment-heavy headers typical of game scripts -- execute the comment lines in constant time per line instead of time proportional to comment length.

Line search in findline (MMBasic.c). When GOTO 1234 or RESTORE 5000 needs to find a numeric line number, the scanner peeks p[T_NEWLINE_HDR] for T_LINENBR; if the peek does not match the target, it advances p += skip and skips the entire line in one step instead of scanning each token. On a program with thousands of un-numbered lines between numbered targets this collapses the find to a header walk.

Label hashing in hashlabels (MMBasic.c, RP2350 builds). PrepareProgram builds the label hash by walking every line and inspecting only the first non-newline / non-line-number token. If that token is not T_LABEL, the loop jumps straight to the next line via the skip byte -- typical programs declare labels on a small minority of lines, so most of the walk is now header-only.

PrepareProgram structural scans (label table, IF/ELSE jump table, sub/function registration). All per-line skip

operations in Commands.c, Draw.c, and FileIO.c that previously called GetNextCommand (which re-tokenises every byte of the line) now p += T_NEWLINE_HDR after the header and p += skip to reach the next header when the line is not interesting. The single-pass program preparation described in the next subsection is what makes these scans fast enough to run unconditionally on every RUN / LOAD / EDIT.

Per-line copy in flash writers. Because the skip byte is non-zero (range 3..0xFE) and is stored at offset +1 from T_NEWLINE, per-line copy loops that stop at two consecutive zero bytes never see a false terminator inside the header itself. Empty lines (just T_NEWLINE + content 0) are still safely transferred because the skip byte holds the line length and the terminating 0,0 pair lies past the content.

The optimisation is independent of OPTION TRACECACHE and OPTION PROFILING: the skip byte is part of the program-memory layout and is built whether or not either feature is enabled.

Optimisations Active in All Builds

The following optimisations are compiled into every build variant -- they do not require the CACHE flag and are not gated by any OPTION setting. They apply equally to PicoMite, PicoMiteVGA, PicoMiteWeb, PicoMiteUSB, and HDMI builds on both RP2040 and RP2350.

IF / ELSEIF / ELSE / ENDIF Jump Table

Every multi-line IF block in the loaded program (and in the library, if one is loaded) is indexed by PrepareProgram into a sorted table of token-address entries. Each entry records:

- the address of the IF / ELSEIF / ELSE token,
- the address of the next sibling arm (the next ELSEIF, ELSE, or ENDIF),
- the address of the matching ENDIF,
- the source line pointer for error reporting.

At run time, cmd_if and cmd_else perform an O(log N) binary search to jump straight to the next arm or past the ENDIF. The previous behaviour was a linear GetNextCommand walk that re-tokenised every intervening statement on every false branch -- so the speedup is largest in deeply-nested IF blocks and in IFs that wrap large bodies (typical state machines, menu dispatchers, etc.).

Single-line IF expr THEN cmd [ELSE cmd] is not entered into the table; it is resolved inline from the parsed argument list with no scan at all.

If the table cannot be built (allocation failure, IF nesting deeper than 64, unmatched ELSE/ENDIF) the affected entries are simply left empty and the run-time falls back to the original linear scan, preserving the original error-reporting behaviour. Library code and program code occupy separate sorted slices of the same table so a binary search picks the correct slice based on which memory region the IF token lives in.

The table is rebuilt automatically by RUN, NEW, LOAD, EDIT, and any other operation that calls PrepareProgram. It is freed when the program is cleared.

FOR / NEXT Direct Pointer Stack

cmd_for performs the matching-NEXT scan exactly once per loop entry and stores the resulting program-memory pointer in the forstack slot, alongside a direct pointer to the loop variable's storage. Subsequent iterations therefore:

- look up the loop variable in O(1) by dereferencing the saved pointer (no name hash, no findvar),
- compare pointers -- not strings -- between cmdline and the saved nextptr to identify the matching FOR slot when cmd_next runs,
- apply the cached integer or float STEP and the cached TO limit with a single load and compare.

This is what makes a tight For i = 0 To N loop faster than the equivalent Do ... Loop Until even on builds without the trace cache. Multi-variable NEXT i, j reuses the same pointer chain to close several loops in one statement without rescanning. The STEP value, TO value, variable type, and LocalIndex are all captured at FOR entry, so changing the step variable inside the loop has no effect (matching standard BASIC semantics).

g_forindex provides up to MAXFORLOOPS nested levels. Re-entering the same loop variable (e.g. via a GOTO out of the loop) safely removes the stale stack entry before pushing a new one.

GOTO / GOSUB Line and Label Lookup

Numeric line targets and label targets used by GOTO and GOSUB are resolved by findline / findlabel. On CACHE builds the resolved target is cached via TCF_JUMP; on non-CACHE builds the lookup runs every time but is itself optimised -- labels are stored in a per-program table built by PrepareProgram and located by binary search rather than by re-tokenising the source.

`commandtbl_decode` / Token-Indexed Dispatch

Every command in ProgMemory is stored as a packed CommandToken rather than a string. ExecuteProgram decodes the token in one indexed table load and dispatches via a switch on the token value. This applies to every statement in every build and is the reason MMBasic can run at the speeds it does without any caching layer at all.

Single-Pass Program Preparation

PrepareProgram walks the loaded program once and in that single pass:

- builds the IF/ELSE jump table described above,
- builds the label table used by GOTO/GOSUB,
- pre-tokenises all commands, function names, and operators,
- validates SUB/END SUB, FUNCTION/END FUNCTION, and TYPE/END TYPE pairing,
- registers each SUB/FUNCTION in the call-target table for O(1) dispatch.

Programs therefore start running with all structural lookups already resolved; the interpreter never needs to re-scan source for control-flow targets at run time.