

TI-59 Fast Mode: Running Programs at ROM Speed

What fast mode is

The TI-59 runs some programs that are not in RAM. The built-in functions $\Sigma+$, $\bar{x}$, $P \rightarrow R$, D.MS and Op 11–15 are short keycode programs stored in ROM, and Solid State Software modules hold up to 5000 more steps. TI built a faster way to run these programs. Between two steps the firmware skips work that a user program normally needs:

- **No display update.** X is not formatted for the display after each step. The display stays as it was until the program ends.
- **No R/S check.** The R/S key cannot interrupt the program, and holding GTO shows no intermediate values.
- **No trace printing.** Neither the printer's TRACE switch nor flag 9 prints the steps.
- **No error stop.** An error does not halt the program, even with flag 8 set.

For a ROM routine such as $P \rightarrow R$ this is exactly right: 20–40 internal steps should look like a single operation, with no flicker and no way to interrupt them halfway. The same steps run in about 40 % less time.

The hack

The firmware keeps a *program source* digit that says where the running program comes from: 0 for RAM, 1 for the module, 8 for the ROM routines. The digit is tested in two different ways:

- The run loop asks only "is it non-zero?". If so, it takes the fast path.
- The step fetch tests single bits. Bit 0 means fetch from the module, bit 3 means fetch from ROM, and anything else means fetch from RAM.

The values 2, 4 and 6 are never set by TI's code. They pass the first test but match neither bit of the second. With one of them in place, a user program in RAM runs on the fast path. No normal key writes this digit, so getting such a value there takes a firmware bug. TI-59 users' clubs found and published such bugs.

Rules for fast-mode programs

Allowed:

- Number entry, all arithmetic and parentheses, and all functions.
- STO, RCL, SUM, Prd and Exc, including their INV and Ind forms. Also HIR, CMs, CP, $x \rightleftharpoons t$, CE and CLR.
- St flg, If flg, Fix, Eng, Deg, Rad and Grad.
- Op 00–08, Op 10 and Op 16–39, and Prt, Adv and Pause.
- Transfers to an absolute or Ind address: GTO, $x=t$, $x \geq t$, If flg and Dsz. Dsz loops do work, despite the old advice not to use them.

Not allowed. Each of these leaves fast mode and loses the result:

- **Labels as targets:** GTO, SBR, $x=t$, $x\geq t$, If flg or Dsz to a label, and A–E' called from the program. Use absolute addresses instead.
- **Subroutines:** SBR in every form, including `Pgm nn SBR`.
- **Built-in ROM routines:** $\Sigma+$, $\bar{x}$, $P\rightarrow R$, D.MS and Op 11–15, and their INV forms.

Differences you must expect:

- The display is frozen, and R/S cannot stop the program.
- Errors never stop the program, and nothing is traced.
- INV SBR with nothing to return to does not stop the program.

Ending the program. This is where most fast-mode programs go wrong. The program must end with RST or R/S, and at that moment X must be 0, a number still being typed, or a value just shown by Pause or Prt. If X holds any other result, the calculator hangs with a dark display, and only switching it off helps. Switching off erases the program.

- Safe endings: `CLR RST`, `0 RST`, `RCL nn Pause RST`.
- Hanging endings: `RCL nn RST`, `... = RST`, `RCL nn R/S`.
- A good pattern: store the result, end with `CLR RST`, and let a normal-mode `RCL nn R/S` at step 000 show it.
- `RCL nn Pause R/S` stops with the result on the display, but the calculator is still in fast mode. Pressing any operation key quietly continues the program. Press RST first.

Ways in

All methods put $X = 1.000000000002$ (or ...04, ...06) on the display first; `1 + 2 EE 12 +/- =` produces it exactly.

- **SST** (shortest; [fastmode-sst.ti59](#)). Put a St flg step at step 007 (the last step of the first 8-step block; see "How the hack works") and start the real program at step 008. Key `GTO 007 SST 0 ST0`.
- **St flg at the end of the partition** (TI PPC Notes V9N3P5; [fastmode.ti59](#)). The program ends with St flg on the last step and halts with a flashing display. Key `7 EE`.
- **Acosta's "hi2"** ([calendar-08-acosta.ti59](#)). The trigger is built into the program with a manufactured pseudo-keycode, so a single key starts it.
- **LRN inside a module program** (`Pgm 02 SBR 239`; [FastMode_Zero4.ti59](#)). This is an unrelated bug that reaches the same state.

Flags and Fix after entry

Entry overwrites all ten user flags and the Fix setting. Set any flags you need inside the fast-mode program.

- **Entry from X** (all methods except `Pgm 02 SBR 239`). The first five digits of X set the flag pairs 4/9, 3/8, 2/7, 1/6 and 0/5. A digit of 1, 5 or 9 sets the lower flag, 2 or 6 the upper one, 3 or 7 both, and 0, 4 or 8 neither. So `1 + 2 EE 12 +/- =` sets flag 4, which has no special meaning; `4 + 2 EE 12 +/- =` works the same and leaves all flags clear. A positive X gives floating display.

- **Entry with Pgm 02 SBR 239.** The flags come from steps 013–015 instead, two digits per step from step 015 down. Zeros there clear all flags; 0p 00 at 014–015 sets flags 7 and 1 (TI PPC Notes V6N8P3, and Claborn's PPX article in the Fast Mode Summaries). Fix ends up at 0.

How the hack works

You need three facts about the calculator's internals to follow it.

1. **The status register.** The firmware keeps its bookkeeping in one 16-digit scratch register. That register holds the Fix setting, the current program step, the user flags, and the program source digit.
2. **Operations come in pieces.** A key such as St flg needs a second key, the flag number. The firmware stores the half-finished operation, waits for the digit, and then jumps to the code that completes it. It computes that jump target by combining address bits.
3. **HIR registers.** HIR STO n copies X, digit for digit, into internal register n. HIR normally accepts only $n = 1\text{--}8$. Register 0 would be the status register.

The bug. Run from a program, St flg expects a two-digit flag number. If a key such as STO, EE or $x \rightleftharpoons t$ arrives after the first digit, the firmware ends the operation early. But the code that decoded that key leaves one stray address bit behind, and St flg's completion adds its own bits on top of it. The jump lands in the wrong routine. After one more misrouted jump, execution ends in the code of HIR STO, with register number 0.

The payload. X is now copied into the status register, digit for digit. 1.0000000000002 is laid out so that each digit lands in a useful field:

- The final 2 becomes the program source.
- The zeros become step 000 and floating-point display.
- The leading 1 happens to set flag 4.

Why it starts running. Like every operation, the misrouted St flg ends in the display routine. That routine sees a non-zero source, assumes a ROM routine such as $P \rightarrow R$ is between two of its steps, and fetches the next step instead of formatting X. This is TI's own mechanism for running ROM routines; from here on every step takes the fast path, and the fetch still reads RAM.

Why the program starts at step 008. The RAM chips transfer only whole registers of 16 digits. A register is either one data register or 8 program steps (000–007, 008–015, ...). The firmware therefore copies a whole block of 8 steps into a buffer, a shift register with the current step at the bottom. Within a block it only shifts the buffer down by one step; it reads memory only when the step counter moves on to the next block. Each shift moves an empty step (keycode 00) in at the top. Each regular jump (GTO, RST, SBR, BST, label search) marks the buffer stale, so counter and buffer always agree. The next fetch reads the target's block and shifts it down until the target step is at the bottom. After GTO 007, for example, the buffer holds step 007 and seven empty steps above it.

HIR STO knows nothing about the buffer. The hack sets the counter to 000 but leaves the buffer as it was, at the step that held St flg's flag number. Until the counter leaves block 000–007, the two disagree:

- Steps 001–007 run the steps that follow in the old block, then the empty steps that the shifts moved in. Each empty step types a harmless 0.
- The first of these steps is swallowed. With `SUM 06` there, the SUM has no effect and the 06 types a 6.
- Step 008 is the first read from memory. From there on, counter and buffer agree again.

St flg therefore goes on the last step of a block: 007, 015, ... $(8k+7)$. Then nothing is left over: `GT0 007` already filled the buffer with empty steps above St flg, so steps 001–007 are all empty, and the real program starts at 008. With St flg on step 003 and `1 2 3 4` on steps 004–007, the run types 234000 before step 008.

Acosta's "hi2" reads the flag number from the program. With St flg at 135 $(8 \cdot 16 + 7)$, that read already moves into the block 136–143. Steps 001–007 then run 137–143, and Acosta puts `GT0 062` there.

Steps 000–006 are never read during the run, so they can hold anything.

Why the restrictions follow. All the other places that look at the source digit also ask only "non-zero?", and each draws the conclusion that suits a module program:

- **Label search** scans the module, because a module program's labels live there. The RAM label is never found.
- **A subroutine call** saves the module's position as the return address, because that is where a module program continues. The return then jumps into the module.
- **The ROM routines** ($\Sigma+$, $P \rightarrow R$, ...) are called the same way, so they fail the same way.
- **The display** is never formatted. When the program stops, the firmware shows X, but X is still in internal format. The routine that lays out the digits then waits for a condition that never occurs. A typed number, 0, or a value just shown by Pause or Prt is already in display format, so these endings are safe.

Limits

Everything here was observed on Calc-U-59, which runs the original TI-59 ROM unmodified, and not on a real TI-59. Op 09 and card reading or writing were not tested in fast mode.