

Result

Before writing the review I checked the TM1638plus library API (Gavin Lyons' repo) to confirm how `displayText`, `displayBegin`, and the brightness default behave, since several findings depend on that:

- <https://github.com/gavinlyonsrepo/TM1638plus> (source + README; `displayText` shows only the first 8 characters of the string; `displayBegin()` applies the library's default brightness, 5)

1. Summary

This sketch drives a TM1638 8-digit/8-button module as a Dutch-language marquee ("lichtkrant"): it scrolls one of nine stored messages, and the buttons let the user pick a message (1–5), adjust brightness (6), or change scroll speed (7/8), with the LEDs mirroring pressed buttons. The overall structure works and the demo is functional, but there are a few real logic bugs: scroll speed can be driven to zero (making the display unreadable), any button event — even a noise glitch — restarts the current message from the beginning, and the configured startup brightness is never actually applied. Verdict: usable as a demo, needs fixes before classroom use. **Score: 6.5/10**

2. Findings

● High

1. Scroll speed can reach 0 → marquee becomes an unreadable strobe — `LeesKnoppen()`, case 128. The guard is `if(ScrollSnelheid >= 50)`, so at 50 the value becomes 0. In `Lichtkrant()`, `currentMillis - previousMillis_display >= 0` is then *always* true, so the text shifts on every pass through the loop, thousands of times per second.

// case 128:

```
if (ScrollSnelheid > 50) { // was: >= 50 — never let it hit 0
```

```
    ScrollSnelheid = ScrollSnelheid - 50;
```

```
    ...
```

```
}
```

C++

2. Any button event restarts the scroll from the beginning — `Lichtkrant()` / `loop()`. When any button change is seen, `Lichtkrant()` calls `LeesKnoppen()` and returns; `loop()` immediately re-enters `Lichtkrant()`, which rebuilds `BewaarTekst` from position 0. That's intended when the *message* changes (cases 1–5), but it also happens after brightness/speed presses (cases 6–8), and it also happens when a 25 ms bounce or noise glitch reads as a press-then-release inside

LeesKnoppen() (the inner if (Knoppenold != Knoppen) is false, nothing happens, but the scroll still restarts). On a long 66-character message this is very noticeable.

Fix: only restart when the selected text actually changed — pass back whether Welketekst changed, and rebuild the buffer only then:

```
// In Lichtkrant():  
if (Knoppenold != Knoppen) {  
    int oudeTekst = Welketekst;  
    LeesKnoppen();  
    if (Welketekst != oudeTekst) return; // only rebuild for a new message  
    // otherwise: refresh charArray offset or simply continue scrolling  
}
```

C++

(If you keep the current structure, at minimum move the rebuild out of Lichtkrant()'s prologue so a re-entry can resume instead of restarting.)

3. Startup brightness is never applied — setup() / globals. Helderheid = 2 is declared but never sent to the module; tm.displayBegin() applies the library default (5). So the first thing users see doesn't match the "Licht" state shown when they press button 6.

```
void setup(){  
    tm.displayBegin();  
    tm.brightness(Helderheid); // apply the intended startup level  
    delay(50);  
}
```

C++

● Medium

4. Variable-length arrays (VLAs) — Lichtkrant(), LeesKnoppen() (3 places). char charArray[Bewaartekst.length() + 1]; is a GCC extension, not standard C++, and the size comes straight from user-editable strings — a very long text added later silently eats stack. Use a fixed buffer with a compile-time check:

```
#define MAX_TEKST 100  
char charArray[MAX_TEKST];
```

```
String BewaarTekst = "    ";

BewaarTekst += myStrings[Welketekst];

if (BewaarTekst.length() + 1 > sizeof(charArray)) BewaarTekst = BewaarTekst.substring(0,
MAX_TEKST - 1);

BewaarTekst.toCharArray(charArray, sizeof(charArray));

C++
```

5. ~250 bytes of string literals in RAM on a 2 KB ATmega328 — myStrings[]. All nine messages live in SRAM. Combined with the String temporaries in LeesKnoppen(), you're spending a meaningful chunk of the Nano's RAM for no benefit. Store them in flash:

```
const char myStrings0[] PROGMEM = "The Question is... To PIE Or Not To BE ";

// ... etc.

const char* const myStrings[] PROGMEM = {myStrings0, myStrings1, ...};

C++
```

and read them out with strcpy_P/___FlashStringHelper before use. Alternatively, drop String entirely and build the small "Licht N" / "Down NNN" buffers with sprintf — that removes the heap-fragmentation risk too.

6. Unbounded wait-for-release — LeesKnoppen(), while (Knoppen == Knoppenold). A stuck or shorted button hangs the whole sketch forever. Add a timeout:

```
unsigned long start = millis();

while (Knoppen == Knoppenold && millis() - start < 5000) {

    Knoppen = tm.readButtons();

    delay(10);

}
```

C++

● Low

7. "Down 1000" doesn't fit the display — case 64. "Down " + "1000" is 9 characters; displayText shows only the first 8, so at the maximum speed the user sees "Down 100". Use "Dwn " or display just the number.

8. HelderheidOud is dead logic — case 32. Since Helderheid++ happens *before* the comparison and the value wraps mod 4, HelderheidOud != Helderheid is always true. The variable and the check can be deleted.

9. Type consistency — globals `Knoppen`, `Knoppenold`. `tm.readButtons()` returns `uint8_t`; using `int` works but is inconsistent. Also, a combined press (e.g. buttons 1+2 → value 3) falls into switch's default: LEDs light, the sketch waits for release, and the scroll restarts — worth a comment if intentional.

3. What's done well

- The `millis()`-based scroll timing uses the rollover-safe subtraction pattern (`currentMillis - previousMillis >= interval`) — correct even after ~49 days.
- The `memmove(charArray, charArray+1, strlen(charArray))` shift correctly includes the null terminator, and the clear-before-redraw step is genuinely needed near the end of the scroll — this is the part most people get wrong with this display.
- `yield()` in the scroll loop keeps the sketch portable to ESP8266/ESP32.
- The LED mapping loop in `doLEDs()` is compact and correct.
- Comments (in Dutch) explain the *why* of non-obvious steps, like the leading spaces for the right-hand entry and the array-copy rationale.

The core scrolling logic is sound; the fixes needed are the speed-to-zero guard, the scroll-restart behavior, and applying the startup brightness.

GLM 5.3 Flash High