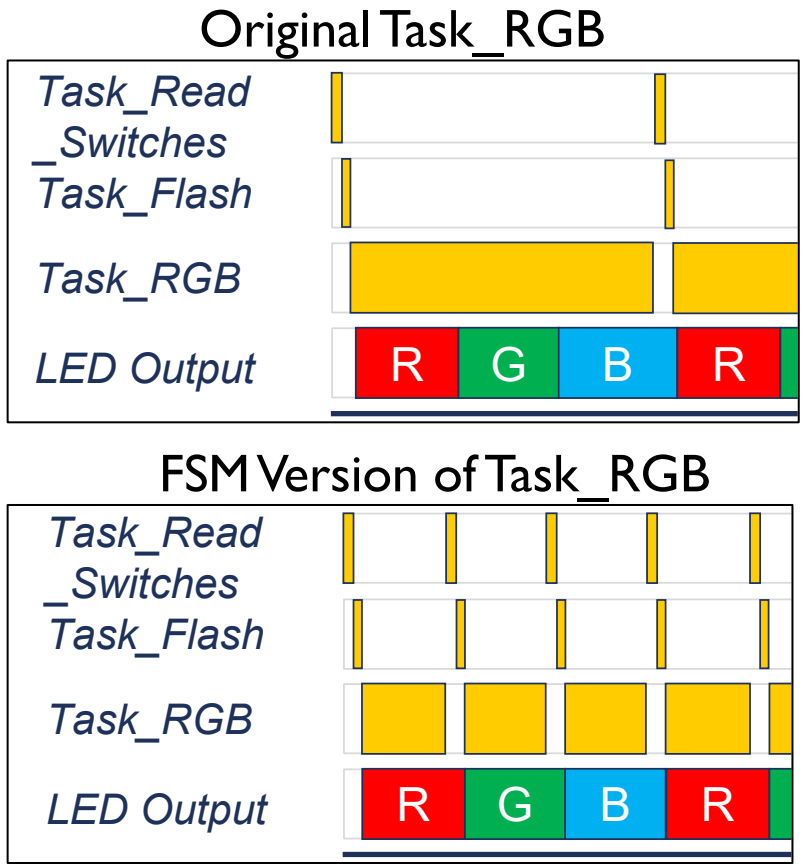
19: Finite State Machines (FSMs) for Interrupt Service Routines and Tasks

v2

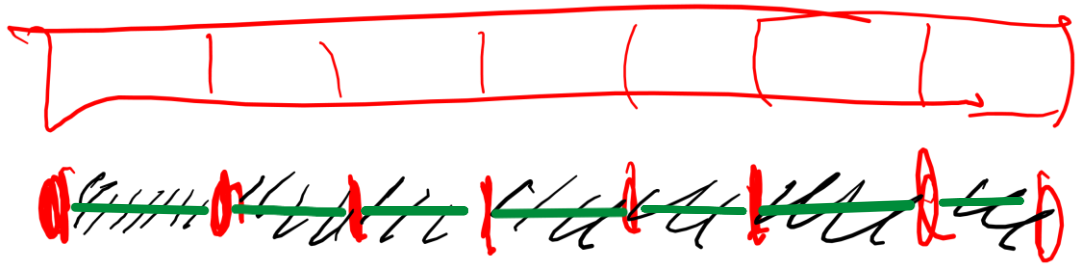
Overview

- Introduction
  - Starting point: **Embedded Systems Fundamentals**: Chapter 3, pp. 65-72
    - Reducing Task Completion Time with Finite State Machines
    - Using Hardware to Save CPU Time
  - Motivation: Share CPU better
    - Better **responsiveness** for other processes: Let another process run sooner
    - **More efficient** use of CPU by this process: Let process pause and share CPU
  - FSM Versions of Tasks
  - ISRs, Tasks and Run-To-Completion processes
- Turning C code into an FSM
  - Concepts
    - Timing budget per call
    - State management concepts
    - Data persistence across FSM activations: static variables
  - Methods
    - Create control flow graph (CFG) for function body source code
    - Identify blocking or slow operations in CFG
    - Group CFG nodes into states which meet timing, control flow requirements
    - Add state management code
  - Synchronization and communication
    - Handling input arguments
    - Handling return value
- FSM Limitations
  - Method impractical for larger, more demanding systems because of poor scaling:
    - Timing analysis
    - Code complexity and transformations
    - Intertask synchronization and communication

Sharing CPU Better

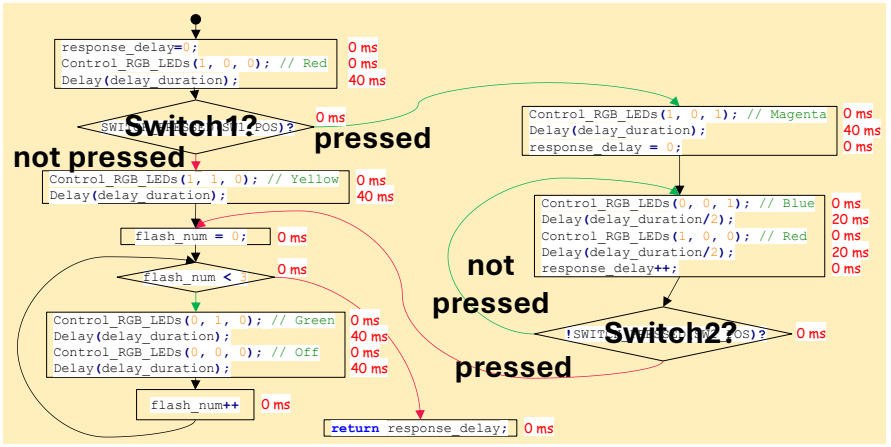


- Better **responsiveness** for other processes:
  - Let another process run sooner
  - Don't do all the work at once, but split it up over multiple runs of the process

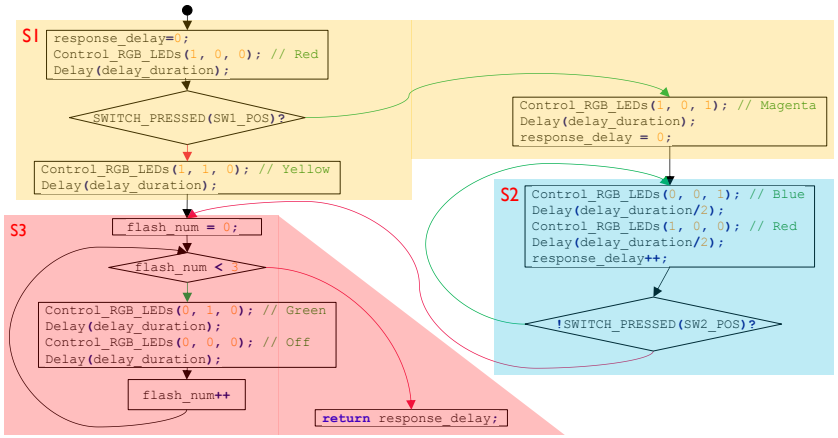


- **More efficient** use of CPU by this process
  - Let process pause and share CPU when appropriate
- Example: I^2C message communication
  - Timing set by protocol's bit rate, can't run as fast as CPU
  - Sequence of software processing steps to interact with hardware at right times
    - send start signal, write byte to Tx, wait, write byte to Tx, wait, write byte to Tx, wait, etc.
  - Protocol's slow bit rate means CPU must wait for a long time between bytes. Busy-waiting (while (!ready);) wastes time
  - FSM conversion will let this process pause after each byte, so scheduler can run other code

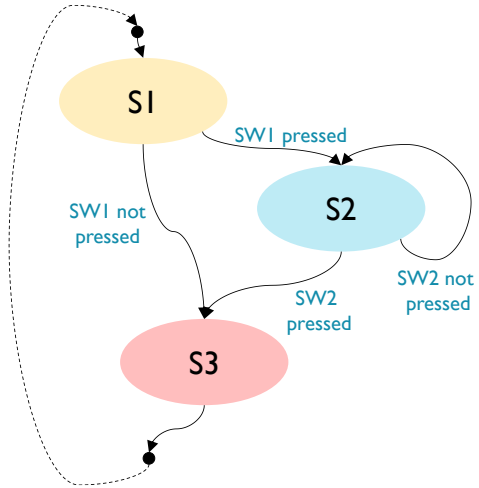
Basic Code and FSM Version



Original Task Code Control Flow Graph (CFG)



Detailed View of FSM version w/ S1, S2, S3



Overview of FSM

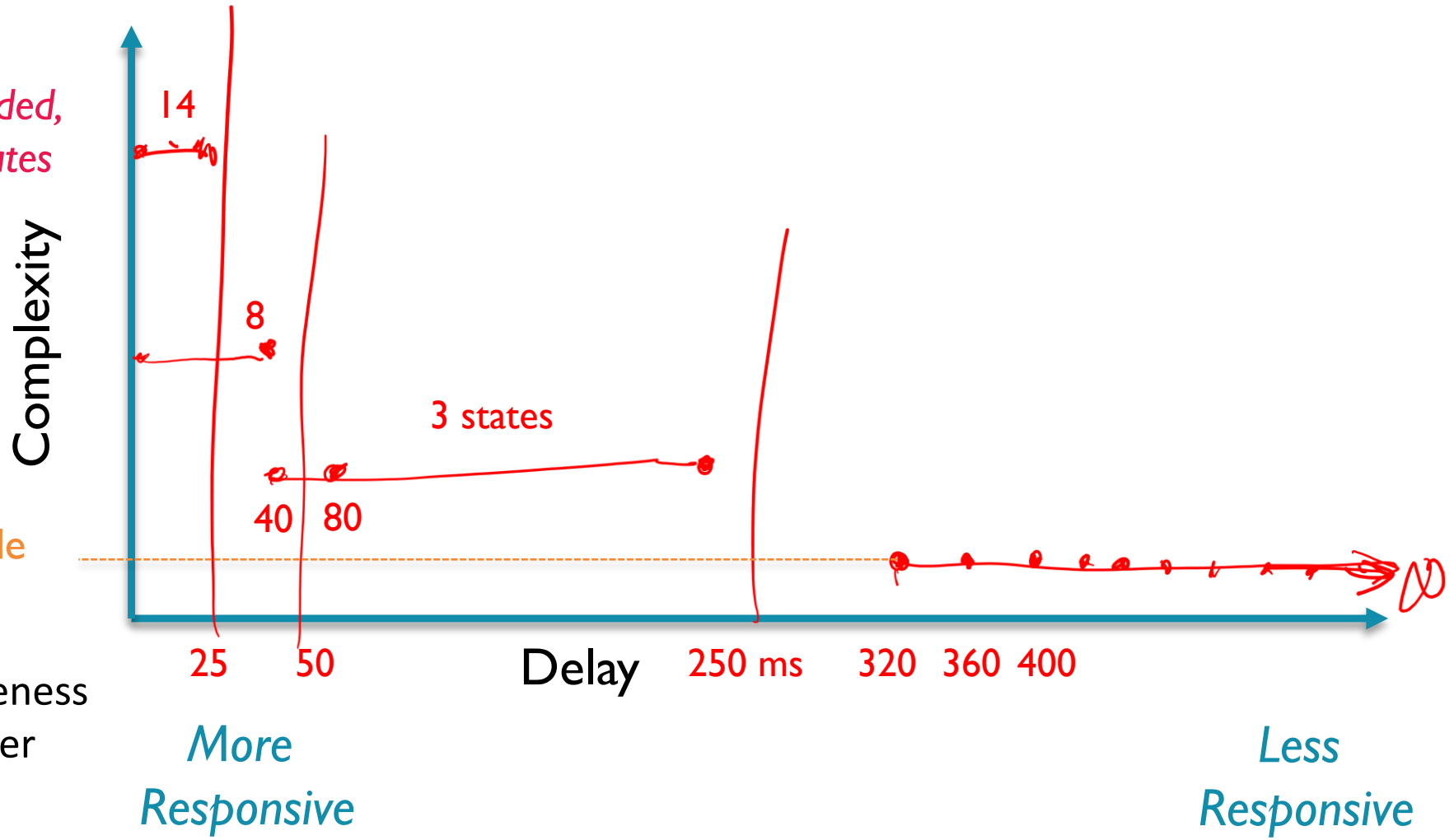
- Example task code flashes LEDs
  - If Switch 1 is pressed, start red/blue loop, don't exit until Switch 2 is pressed.
  - Best case: No switch presses. **Takes 80 ms**
  - Worst case: Only Switch 1 pressed. **Never finishes!**
    - No other tasks can run, just ISRs.
    - (Unless this is an ISR, in which case only higher-priority ISRs can run…)
- Make FSM version of function to do *just some* processing in each call
  - Break function into states which execute within a given **time budget** (e.g. 250 ms)
    - BTW, original code has one state
  - Must run FSM function more times to do same work
- As a result, …
  - Interrupt & task schedulers can run sooner, more often, so …
  - They get finer-grain control of processor's time, so…
  - They can run a different process **sooner** (Benefit #1).

Benefit #2 of FSMs: Better Allocation of CPU Time

- Example: I^2C message communication uses sequence of software processing steps
- Share the processor's time better by **reclaiming idle time** in function for other processing
  - Yield processor during blocking operations (e.g. polling busy-wait loops for hardware status update, etc.)
- Especially useful for synchronization **within SW process** to HW state or event
  - Example: Software must block until HW->Ready == true
  - Converts blocking synchronization operation (loop in SW process: while (!(HW->Ready));) into non-blocking operation,
  - Relies on FSM to allow **RTC** (Run-To-Completion) process to **pause partway through, resume later** with saved context
- Especially useful for interrupt handlers
  - Handlers usually need to be short and fast, but handler processes are **run-to-completion** (no pause & resume provided)

Scaling Limits of FSMs

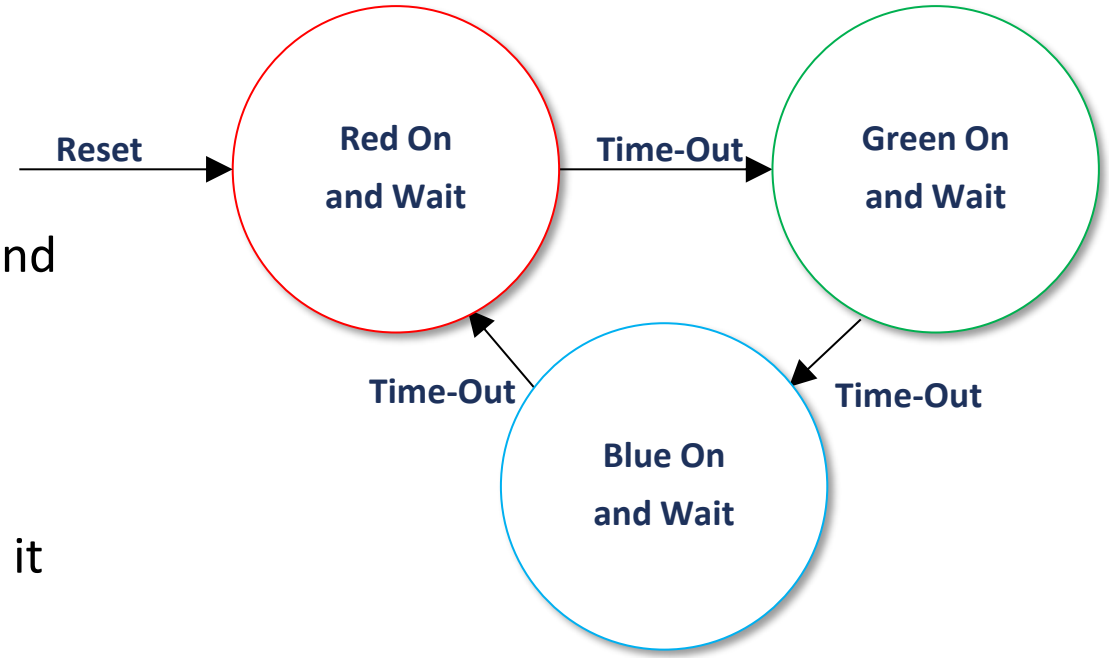
More Difficulty:
More States,
More FSM Calls Needed,
Harder to Create States



- Increasing responsiveness gets harder and harder

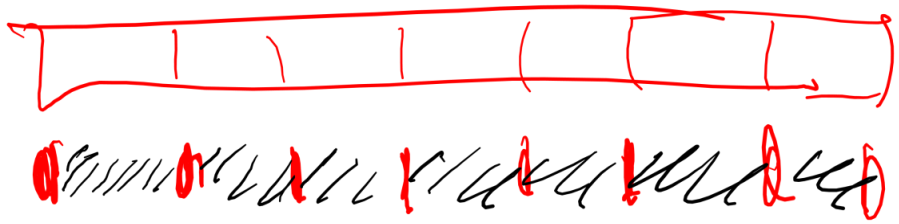
FSM Definitions

- Directed graph
  - Consists of states, directed transitions (arcs, edges) and actions
- Definitions
  - State: the condition the system is in now
  - Next State: the state the FSM will be in the next time it is called (after this state's code finishes executing)
  - Transition: a path between states, usually triggered by an event
  - Action: activity. Can be performed when …
    - … following a specific transition
    - … entering a state, regardless of transition which led to state
    - … exiting a state, regardless of which transition is followed



- Transition fields (all are optional)
  - Event name: what triggers the transition
  - [Guard]: Optional Boolean condition which enables or disables whether event triggers transition
  - /Action list: list of actions to perform on transition

FSMs and ISRs

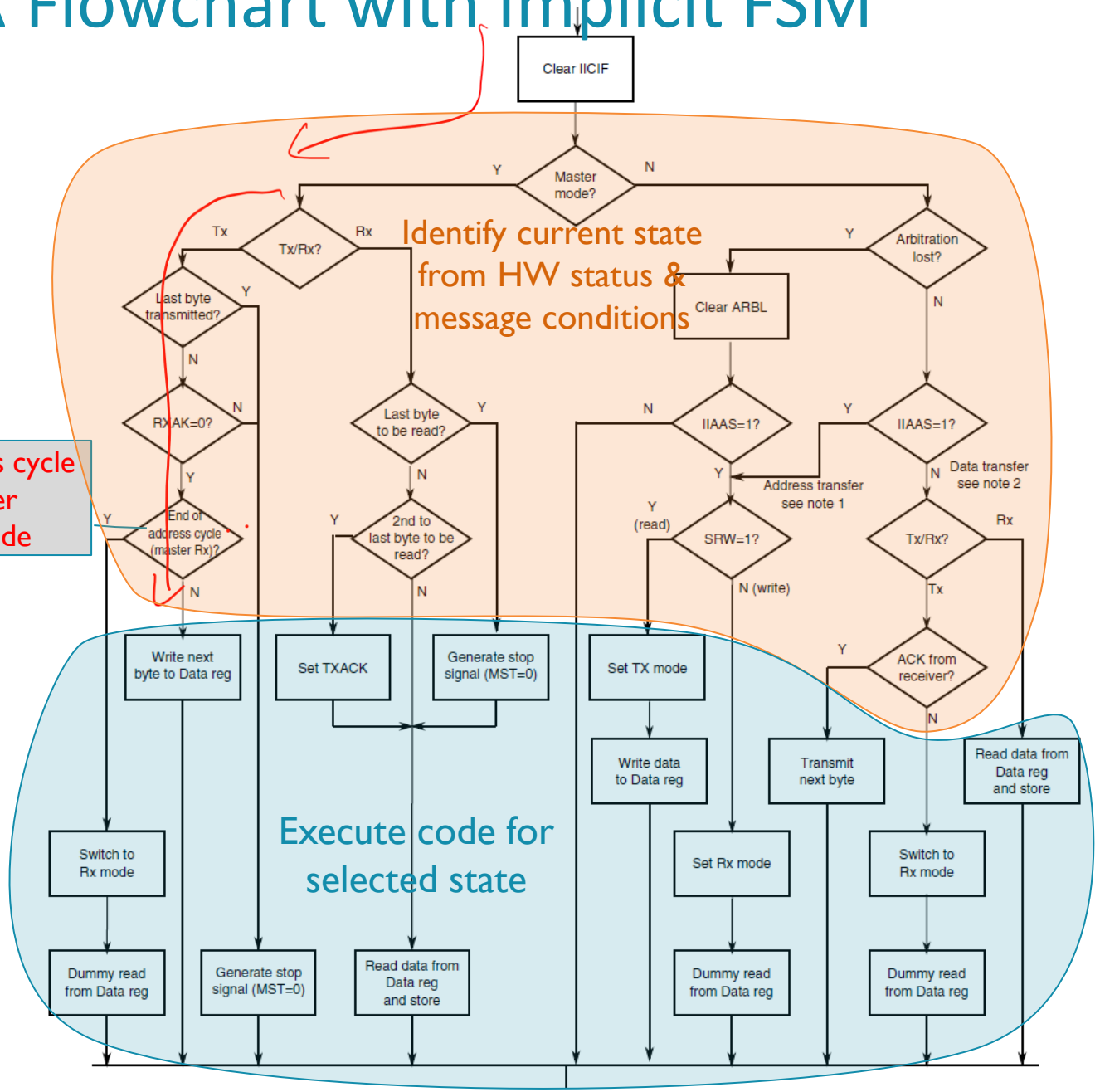


- Example: I2C message communication SW event sequence: send start signal, write byte to Tx, wait, write byte to Tx, wait, write byte to Tx, wait, etc.
- Regular code not good for event-driven processing
  - Must detect when event occurs…
    - Use blocking polling loop. Busy-waiting blocks until event occurs, not sharing CPU.
    - Use interrupt system, but limited scope (details soon)
    - Use OS support (coming soon)
- ISRs are good for processing driven by HW events
  - Event X happened? Then do this processing and generate output Y
  - What if SW processing uses each event to step through chain of sub-processing activities?
- FSMs good for state-based event-driven processing
  - You are in state 3 and got input X? Then do this processing, generate output Y, and go to state 7
- FSMs break up regular code with more *scheduling points,* returning control to interrupt and task schedulers sooner

KL25Z SRM I2C Chapter has ISR Flowchart with Implicit FSM

- I2C ISR
  - An I2C event just occurred? Then do this…
  - Figure 38-42, KL25 SF Reference Manual
- What triggers the interrupt?
  - When in master mode, other code must write first byte to DATA to start transaction. End of byte transmission triggers ISR.
  - Else first byte received

End of address cycle and Master and Rx Mode



Turning C Code into an FSM

Reading Assignment

- **Embedded Systems Fundamentals**: Chapter 3, pp. 65-72
  - Reducing Task Completion Time with Finite State Machines
  - Using Hardware to Save CPU Time

V3 Flasher Code

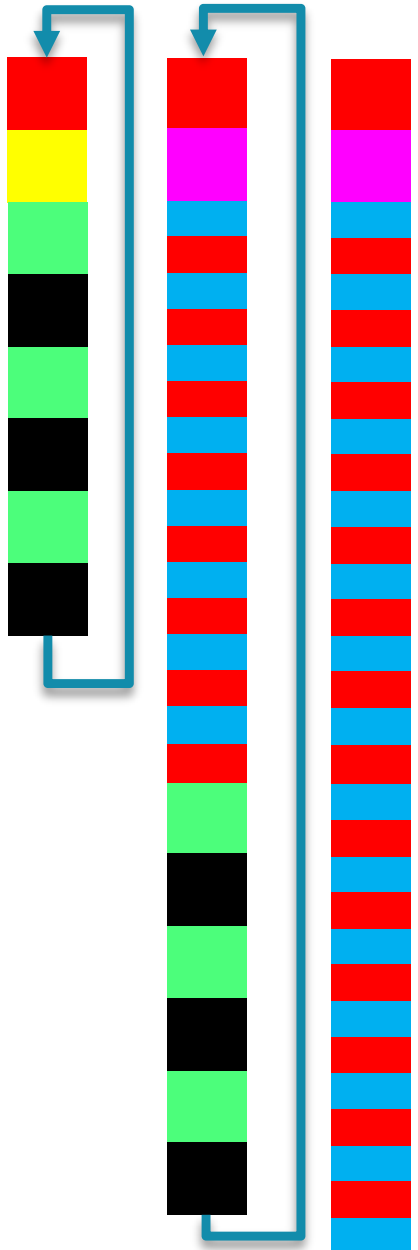
Split long tasks into multiple states

```
void Task_RGB(void) {
    if (g_flash_LED == 0) {          // c
        // Red state
        Control_RGB_LEDs(1, 0, 0);
        Delay(g_RGB_delay);

        // Green state
        Control_RGB_LEDs(0, 1, 0);
        Delay(g_RGB_delay);

        // Blue state
        Control_RGB_LEDs(0, 0, 1);
        Delay(g_RGB_delay);
    }
}
```

```
void Task_RGB_FSM(void) {
    static enum {ST_RED, ST_GREEN, ST_BLUE, ST_OFF} next_state;

    if (g_flash_LED == 0) {      // only run task when NOT in flash
        switch (next_state) {
            case ST_RED:
                Control_RGB_LEDs(1, 0, 0);
                Delay(g_RGB_delay);
                next_state = ST_GREEN;
                break;
            case ST_GREEN:
                Control_RGB_LEDs(0, 1, 0);
                Delay(g_RGB_delay);
                next_state = ST_BLUE;
                break;
            case ST_BLUE:
                Control_RGB_LEDs(0, 0, 1);
                Delay(g_RGB_delay);
                next_state = ST_RED;
                break;
            default:
                next_state = ST_RED;
                break;
        }
    }
}
```

New Example: A *Different* LED Flasher!

- Default: Red, yellow, three green flashes
- Press SW1 to get magenta, then repeating blue/red sequence
  - Press SW2 to stop blue/red sequence
- Code is "non-trivial"
  - Has subroutine calls
  - Has conditional control flow from if/else, do/while loop, for loop
  - One loop has unbounded number of iterations
- How long can task function take?



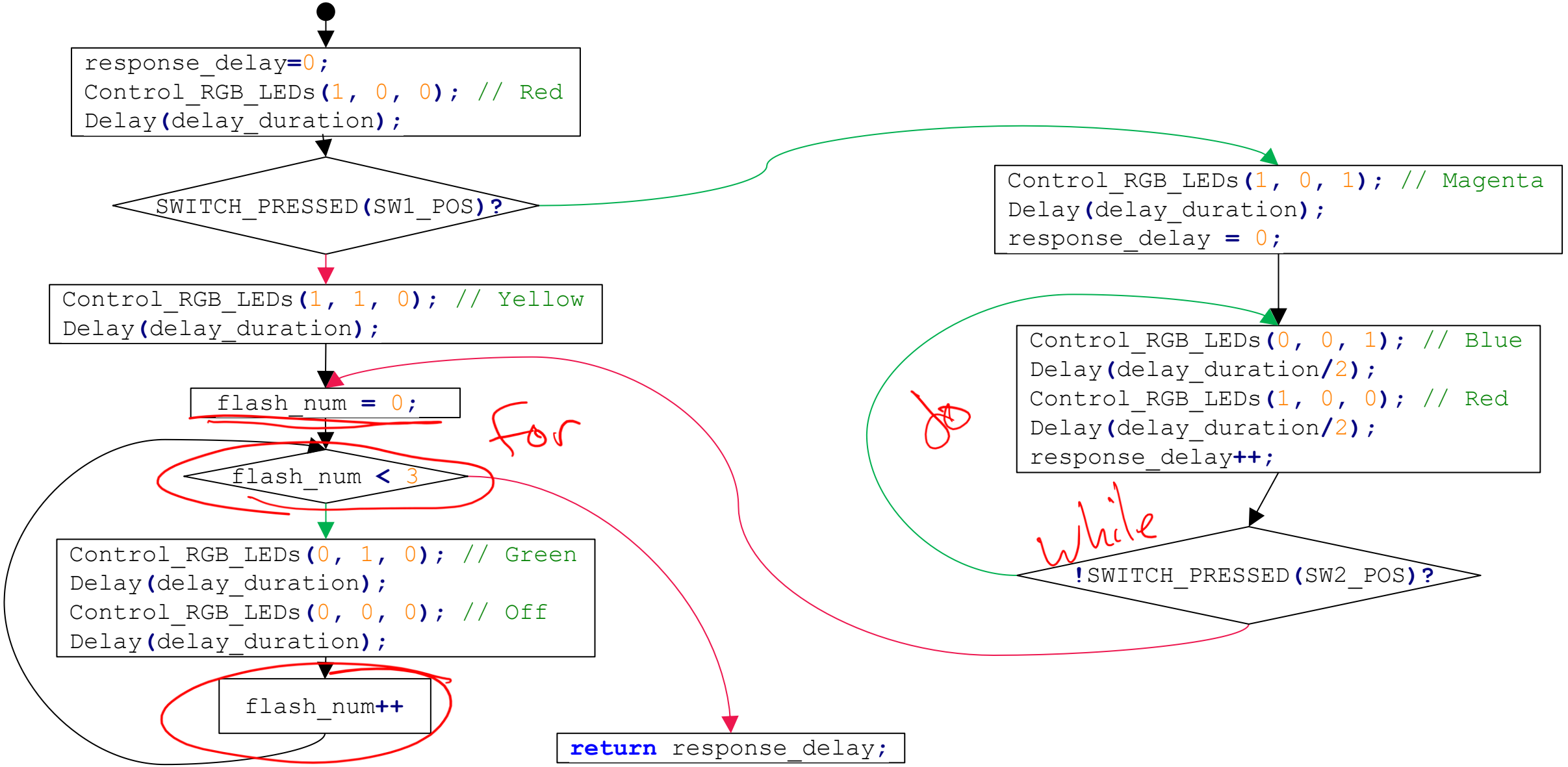
```
int Task_Color_Sequence(unsigned int delay_duration) {
    int flash_num, response_delay=0;
    Control_RGB_LEDs(1, 0, 0); // Red
    Delay(delay_duration);
    if (SWITCH_PRESSED(SW1_POS)) {
        Control_RGB_LEDs(1, 0, 1); // Magenta
        Delay(delay_duration);
        response_delay = 0;
        do {
            Control_RGB_LEDs(0, 0, 1); // Blue
            Delay(delay_duration/2);
            Control_RGB_LEDs(1, 0, 0); // Red
            Delay(delay_duration/2);
            response_delay++;
        } while (!SWITCH_PRESSED(SW2_POS));
    } else {
        Control_RGB_LEDs(1, 1, 0); // Yellow
        Delay(delay_duration);
    }
    for (flash_num = 0; flash_num < 3; flash_num++) {
        Control_RGB_LEDs(0, 1, 0); // Green
        Delay(delay_duration);
        Control_RGB_LEDs(0, 0, 0); // Off
        Delay(delay_duration);
    }
    return response_delay;
}
```

Converting the Function Body

Analyzing the Function

- Create CFG (control flow graph) from C code
  - Essentially a flowchart
- Use as roadmap to split up code into different states

Create Control Flow Graph



Converting the Function Body: Break CFG into States

- Goal
  - Reduce maximum time to execute any state's code to an acceptable value
- Approach
  - Create each state by adding code until
    - Execution time will be too long (delay, blocking operation).
    - Incoming control-flow path from a different state's code
  - Terminate state, add code to update state variable for each exit transitions
  - Repeat
- Basic state formation rules
  - Code has exactly one entry point, but multiple exit points allowed
  - Code must be connected (able to reach all code from the entry point)
  - Don't split state within a subroutine call

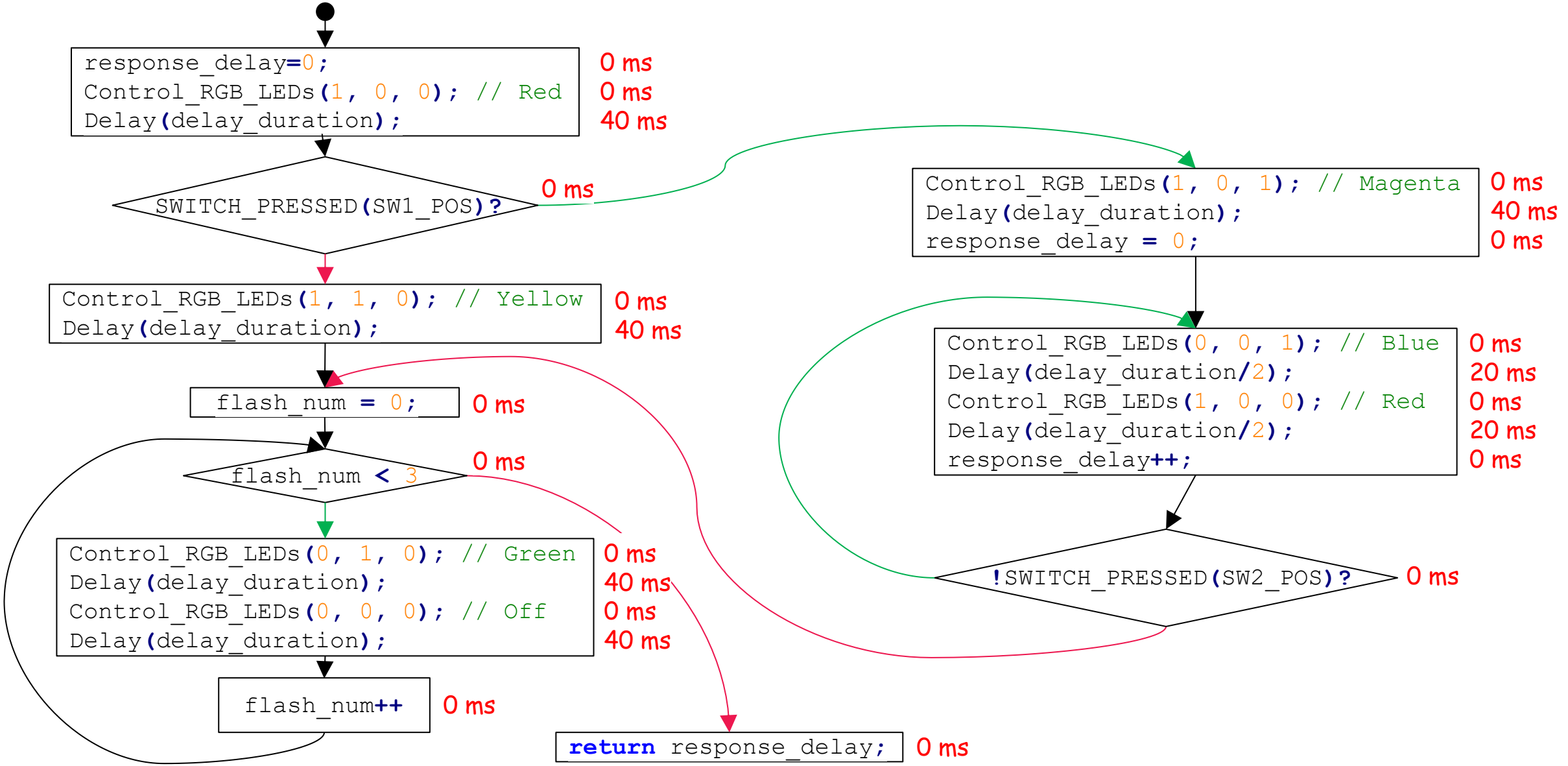
Converting the Function Body: State Management and Statics

- Name the states
- Create static next_state variable as enumerated type, initialize it
- Implement state code selector
  - Add switch statement
    - Insert case statement per state.
    - Update next_state for each transition out of the state
    - Add default case
  - Could also use a table of function pointers to reduce function size
- Convert to static all variables which may be live across FSM calls (i.e. between states)
  - Worst case limit: all local variables

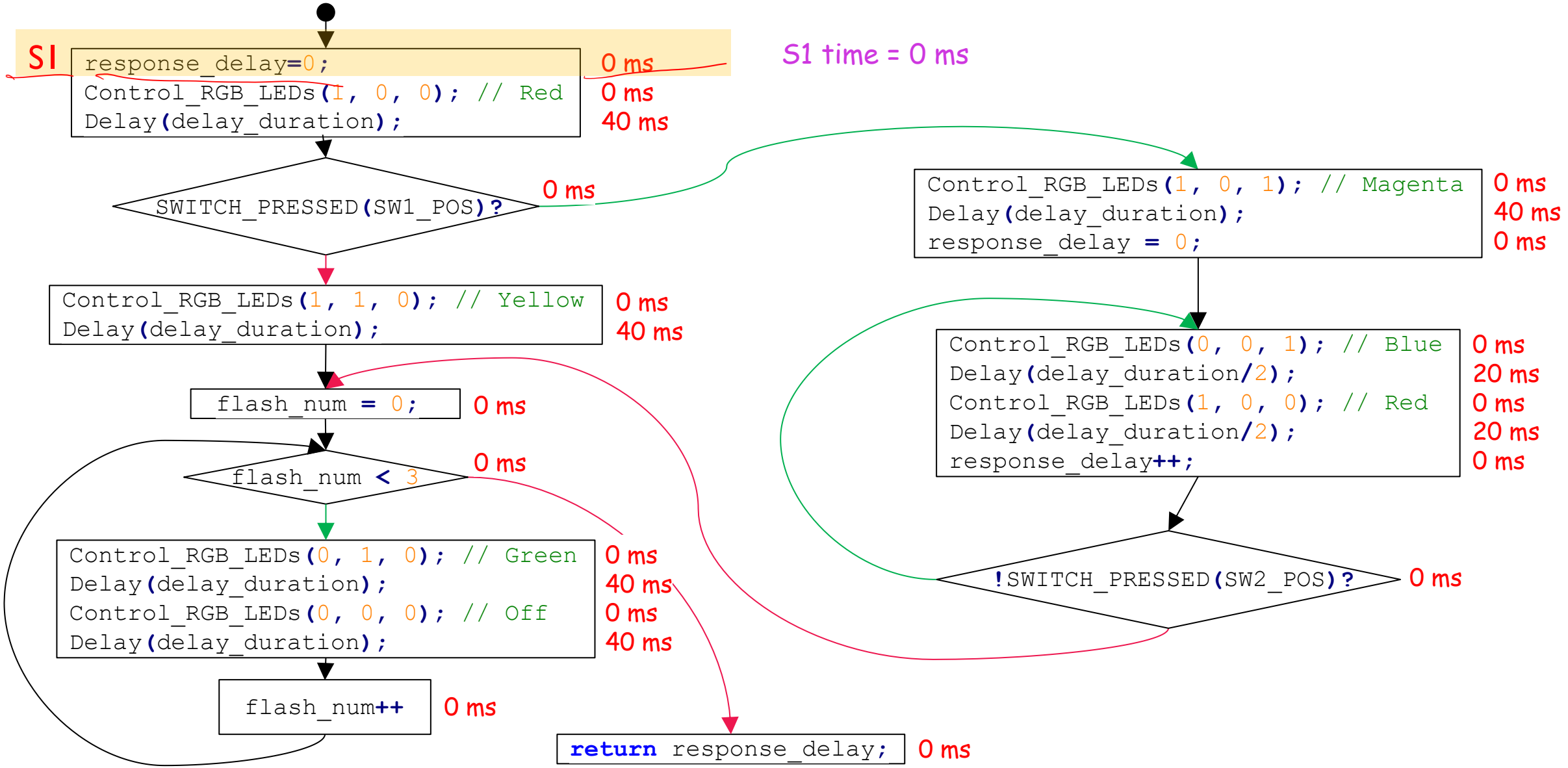
Timing Estimation

- How long (in the worst-case) does it take each line of source code to execute?
- Static
  - Compile and link program. Examine object code (!) and calculate, based on instructions, largest loop iteration counts, longest conditional path
- Dynamic analysis
  - Run code and measure execution time
    - Add debug bits to code, view with logic analyzer
    - Add code to save time stamps from hardware timer
- Use simplifying assumptions here
  - Only Delay calls take any time
  - All other code is zero-time

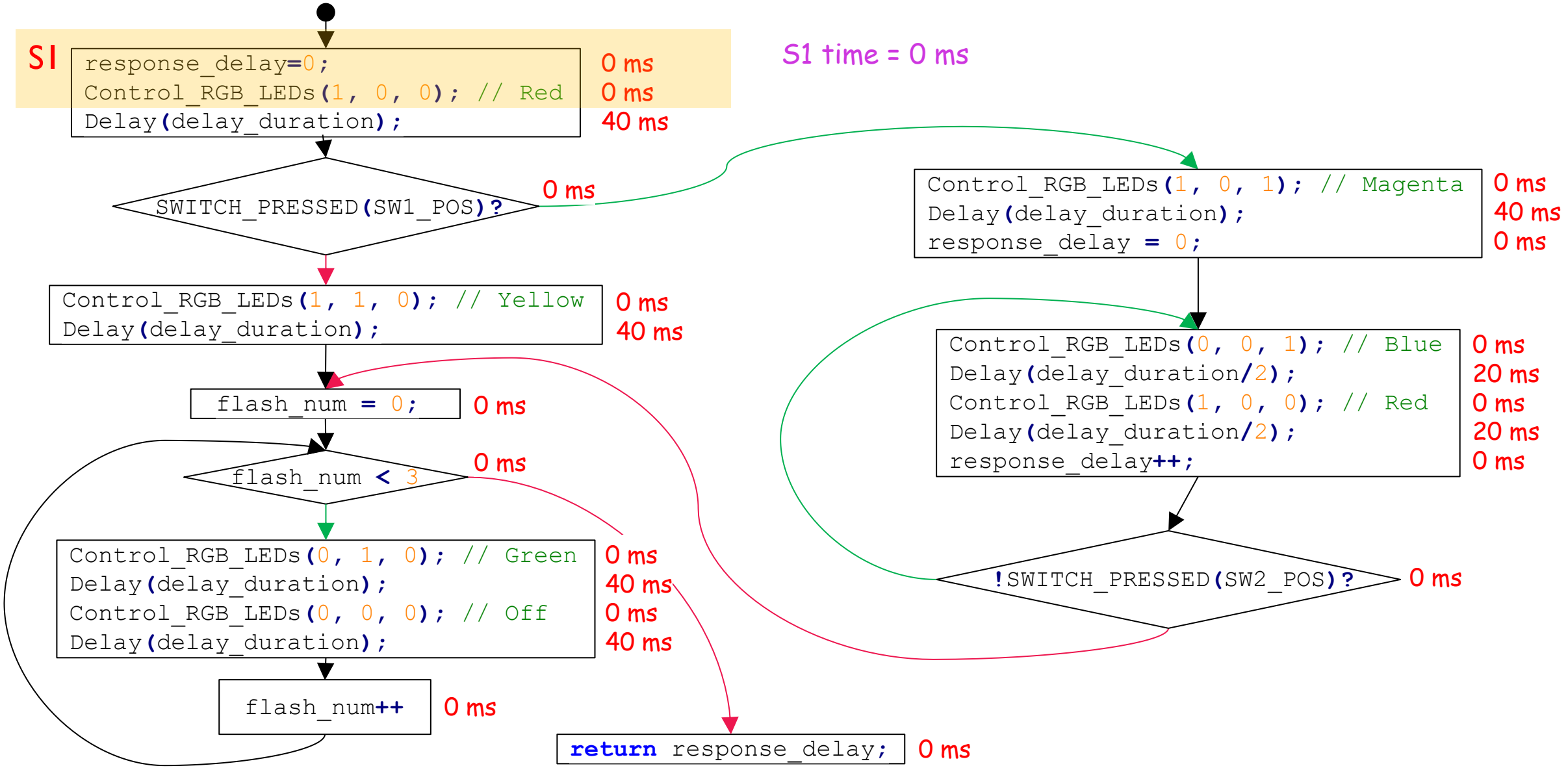
Mark CFG with Approximate Execution Times



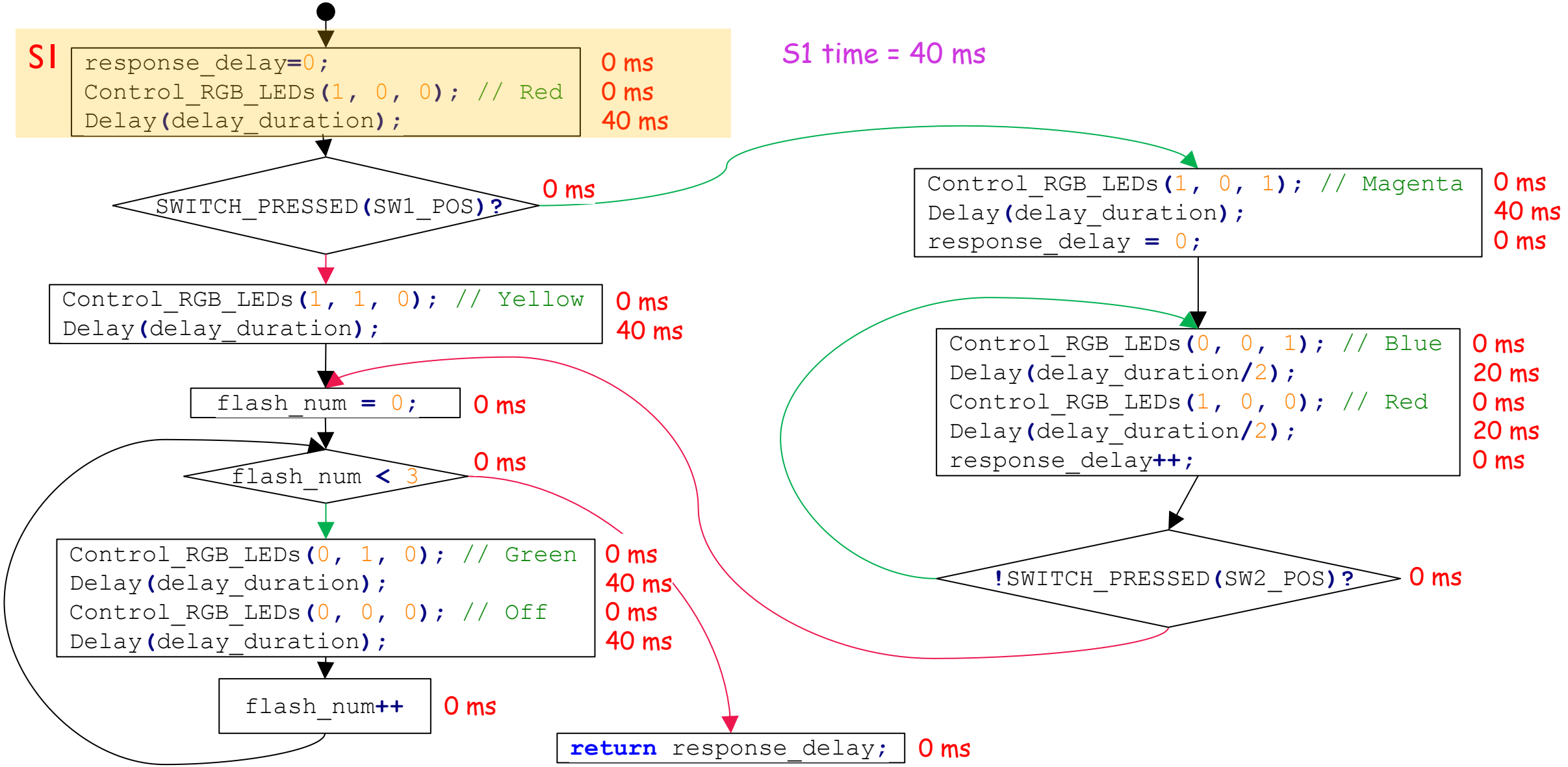
Add Code to State S1 as Allowed by 250 ms Time Budget



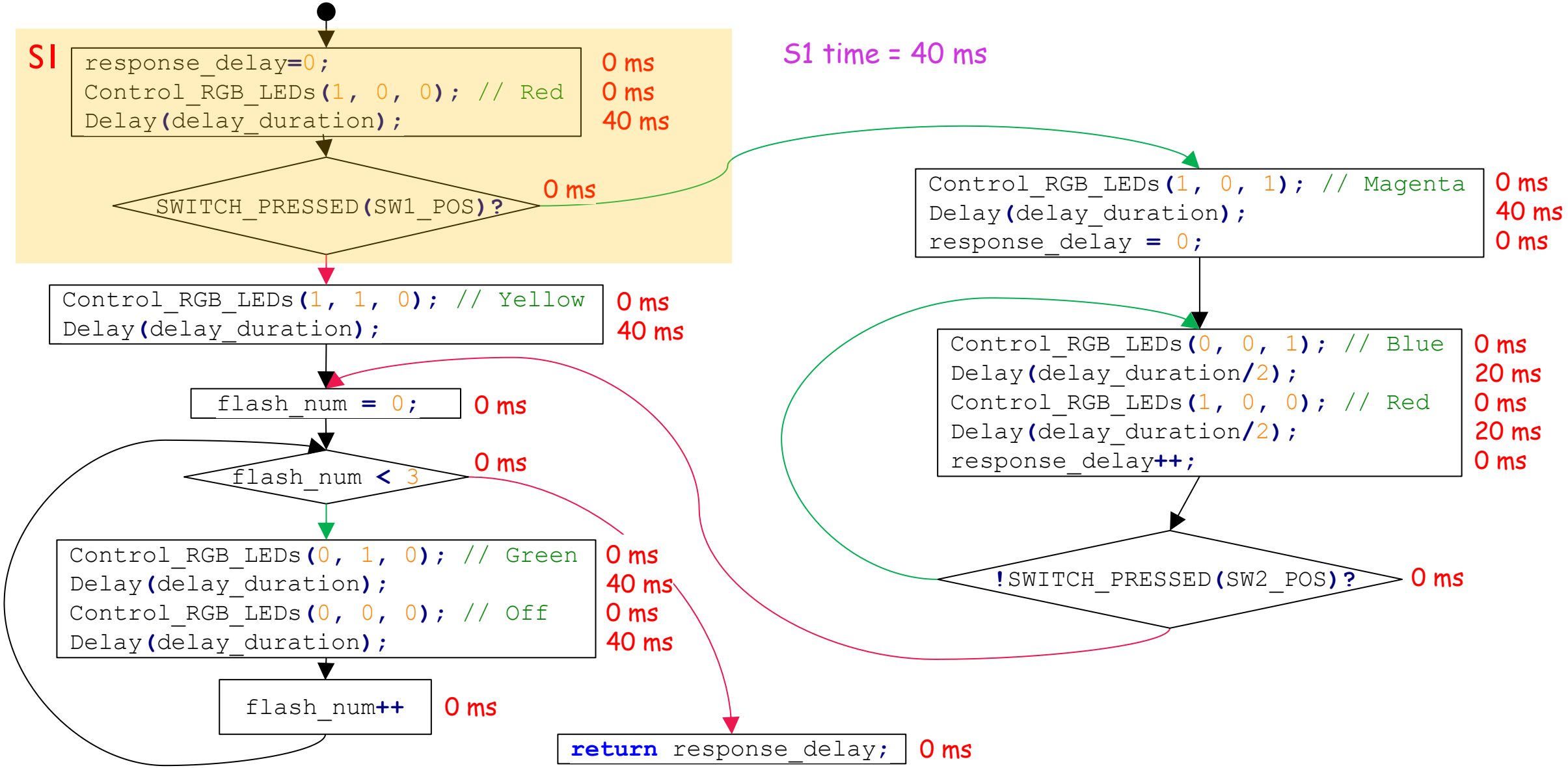
Add Code to State S1 as Allowed by 250 ms Time Budget



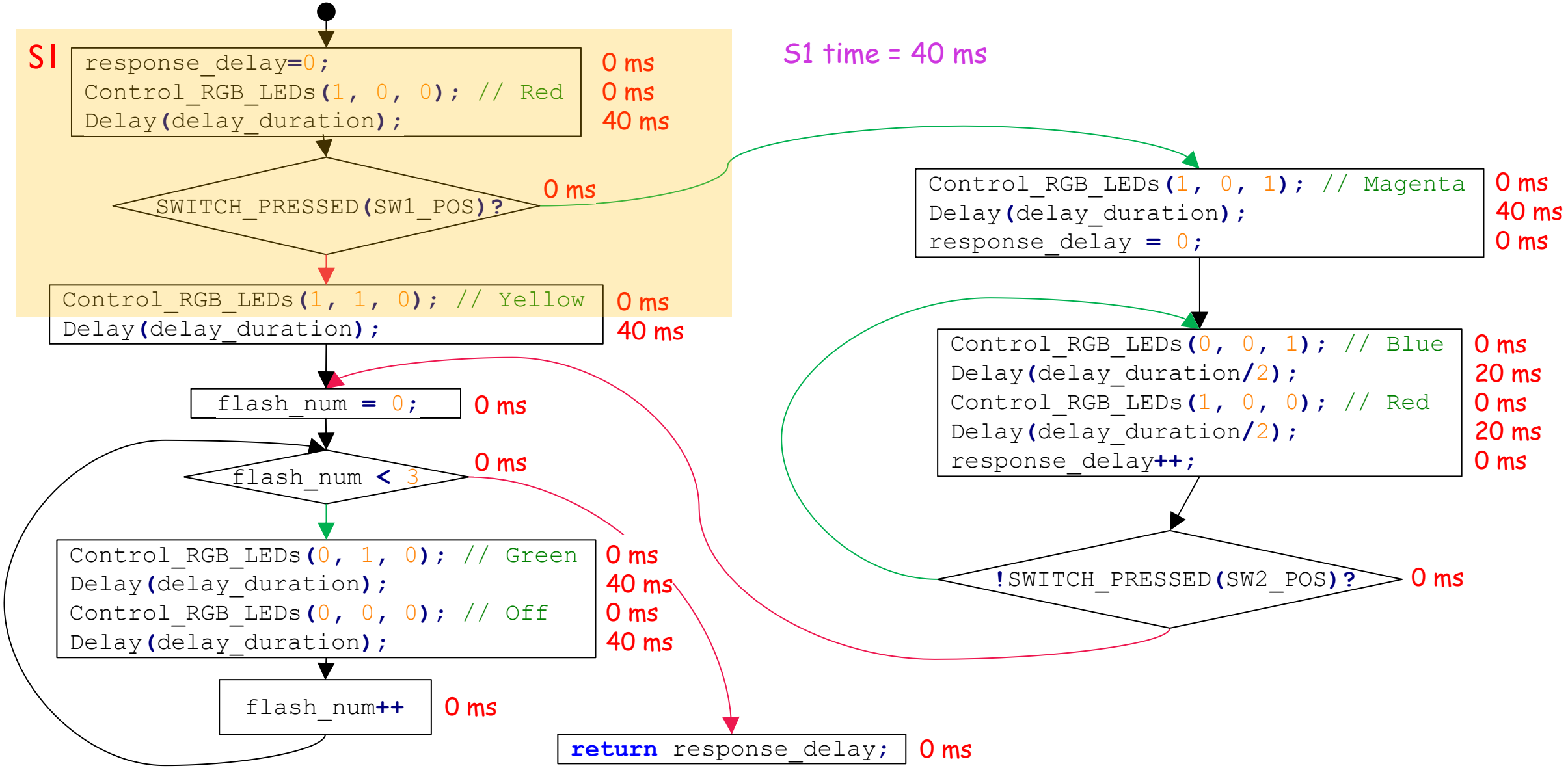
Add Code to State S1 as Allowed by 250 ms Time Budget



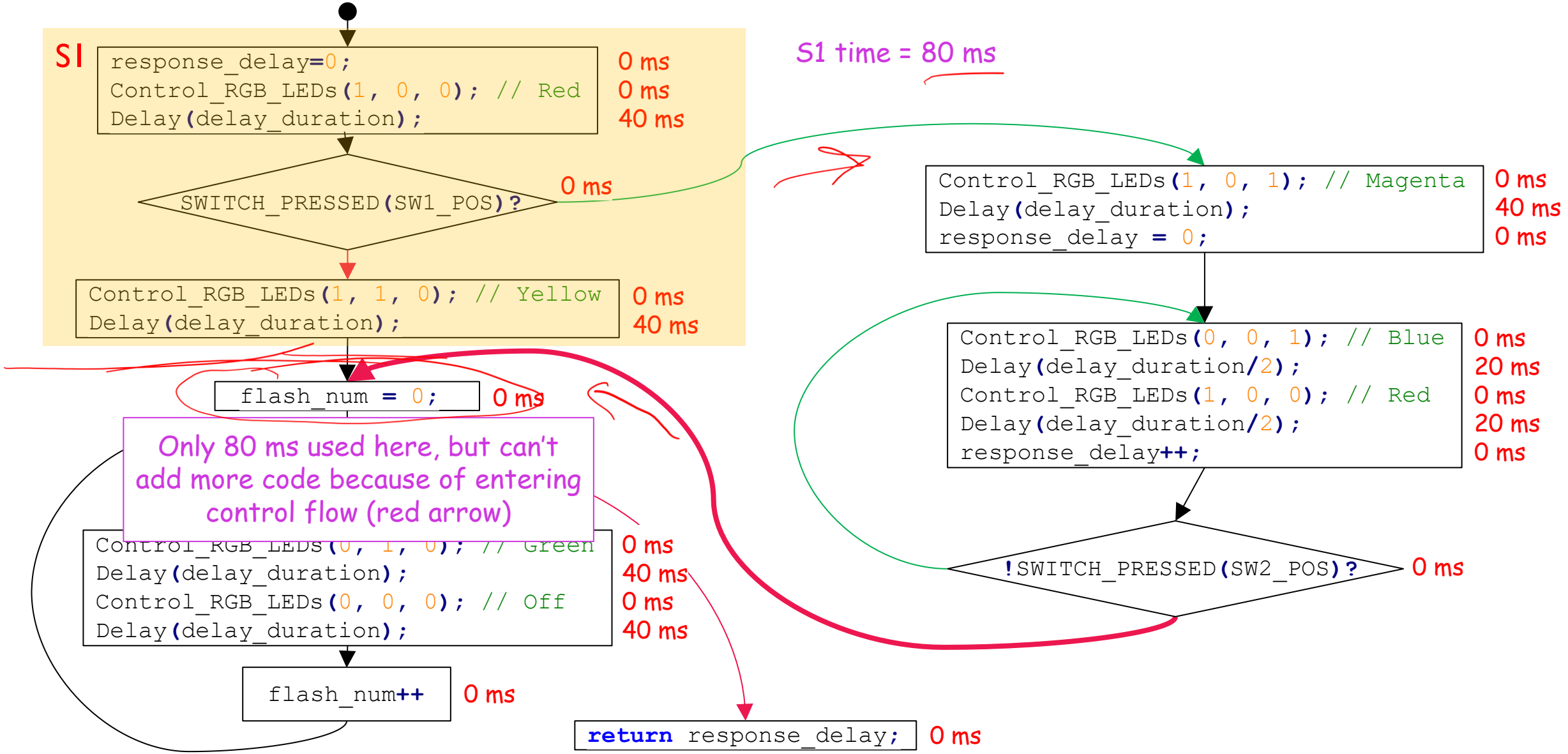
Add Code to State S1 as Allowed by 250 ms Time Budget



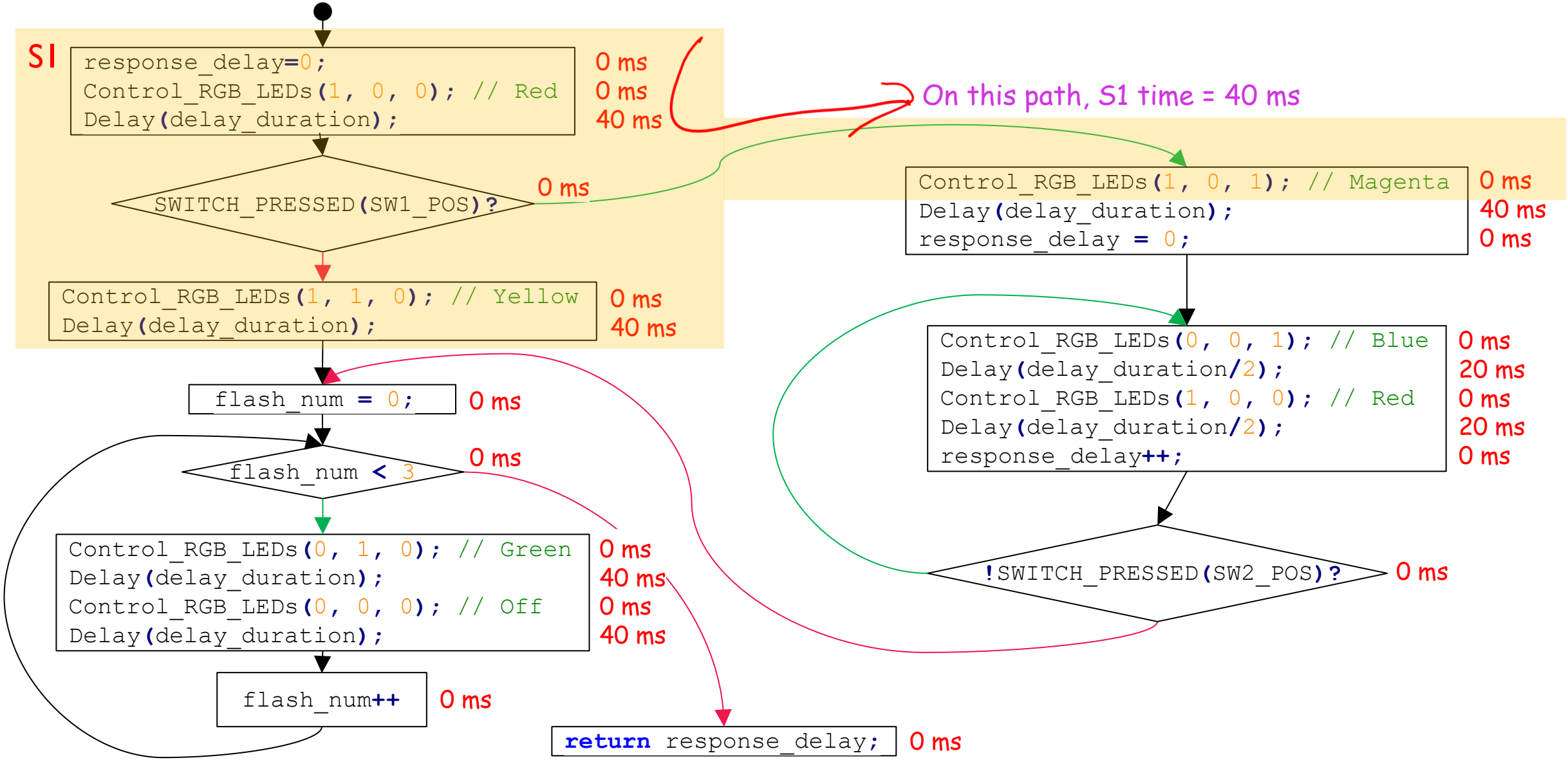
Add Code to State S1 as Allowed by 250 ms Time Budget



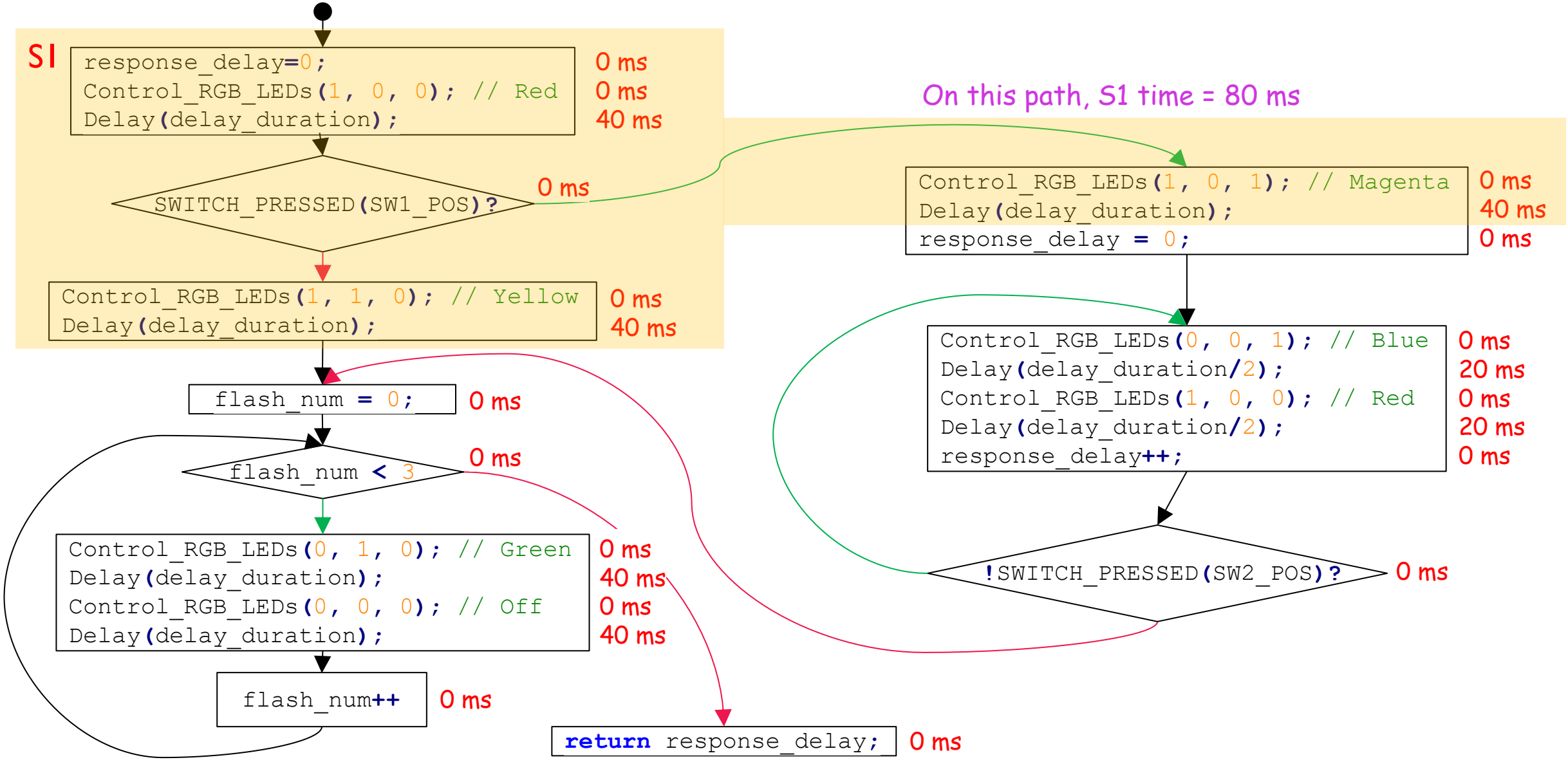
Add Code to State S1 as Allowed by 250 ms Time Budget



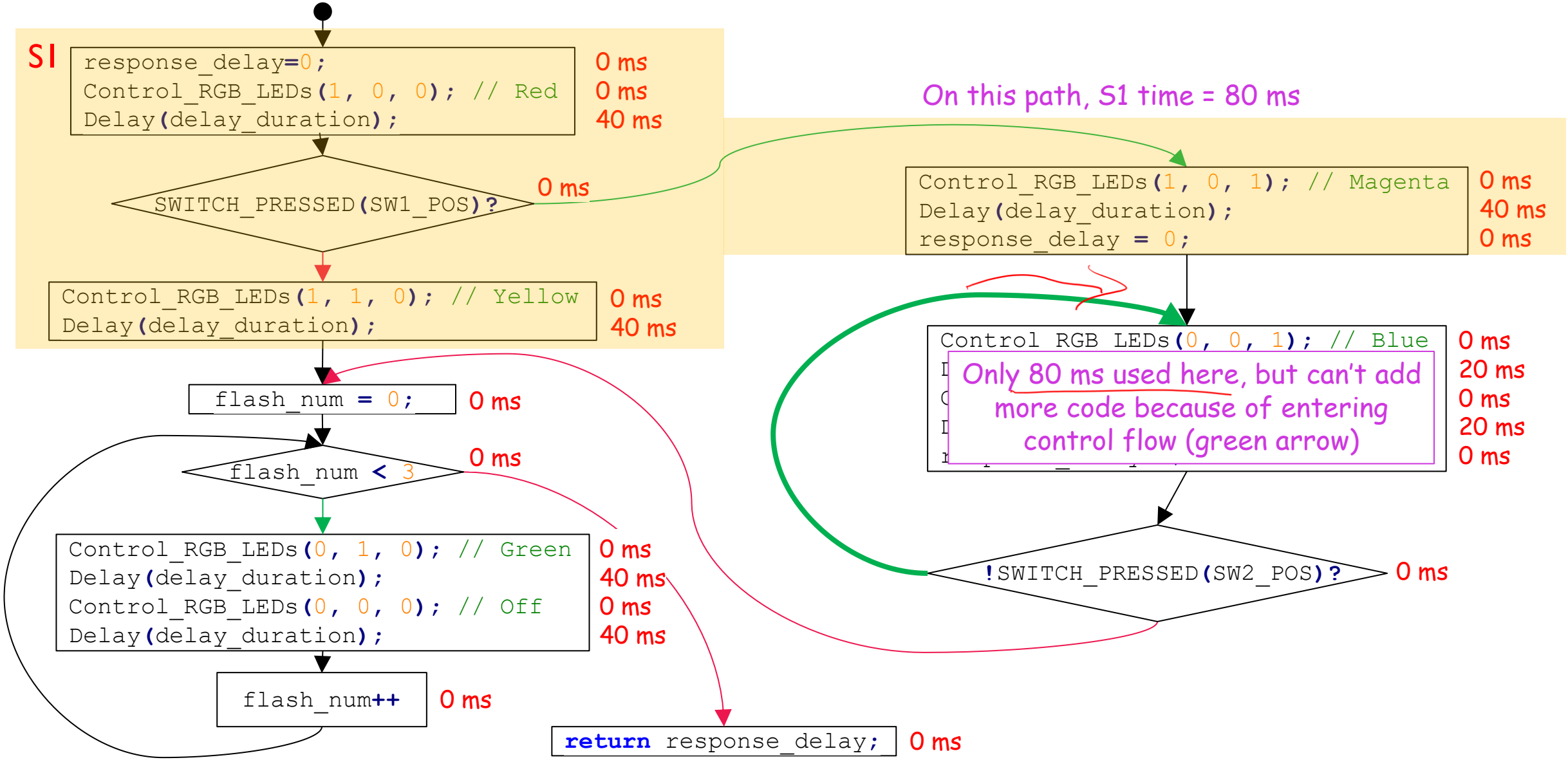
Add Code to State S1 as Allowed by 250 ms Time Budget



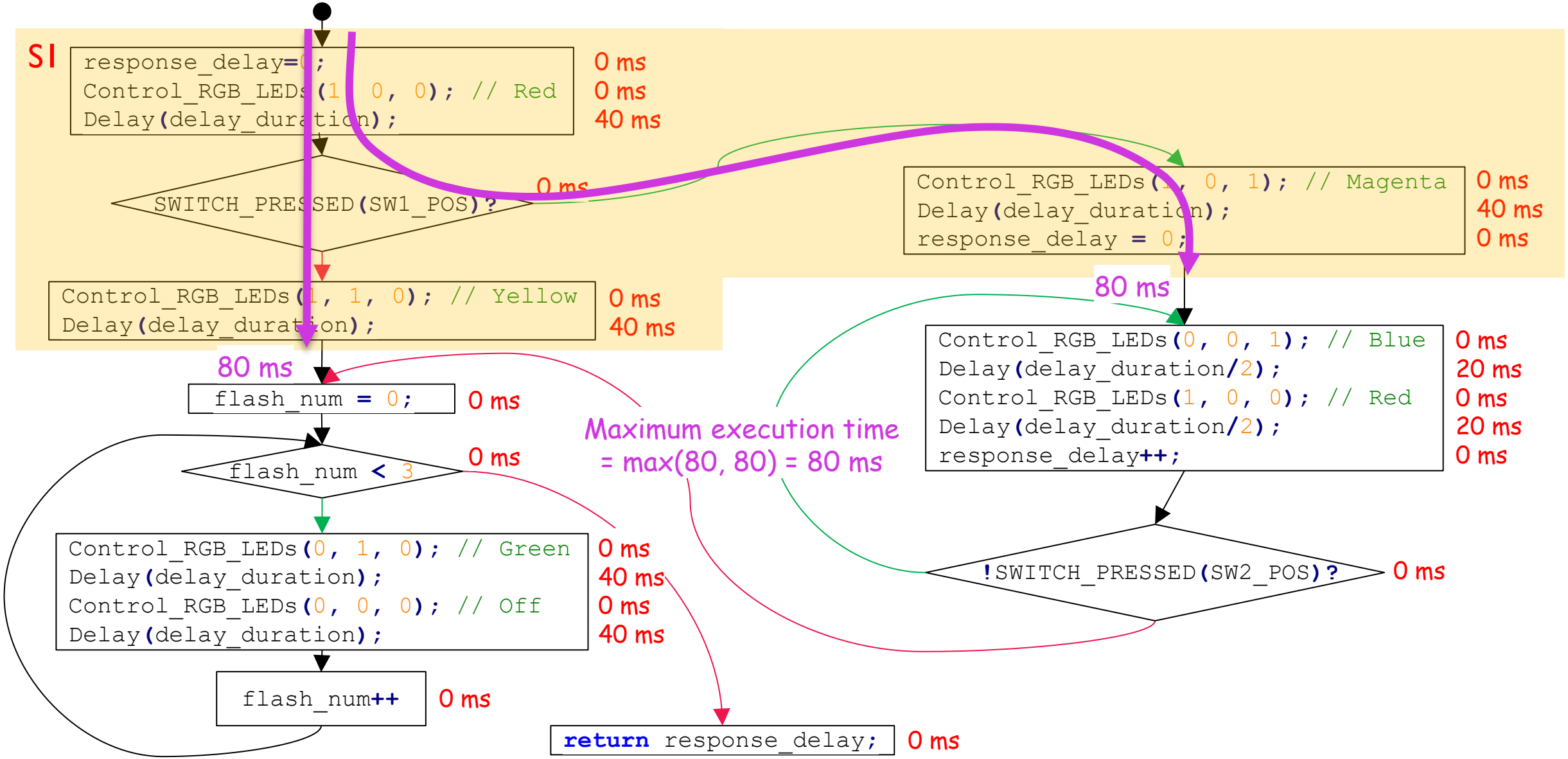
Add Code to State S1 as Allowed by 250 ms Time Budget



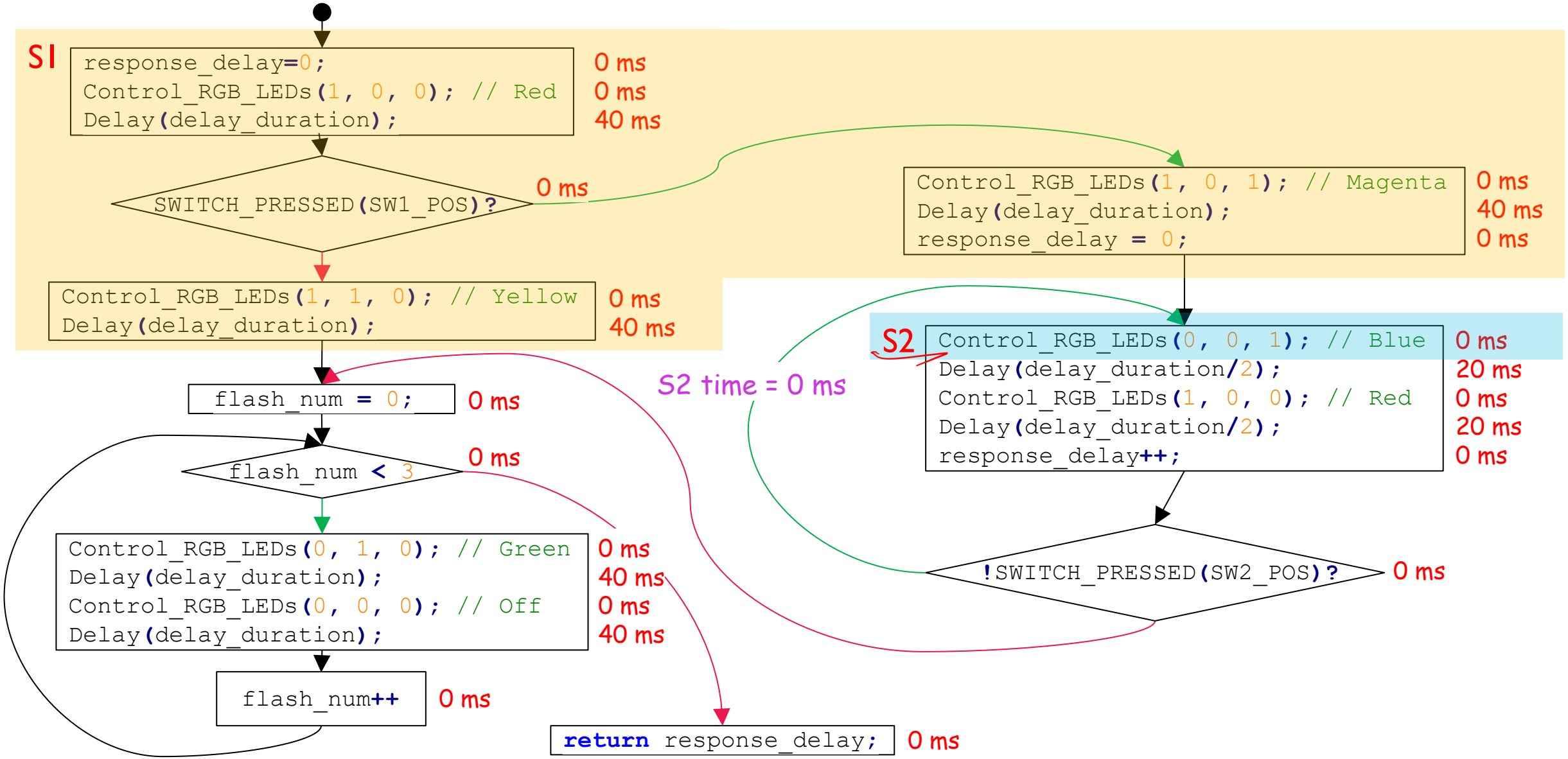
Add Code to State S1 as Allowed by 250 ms Time Budget



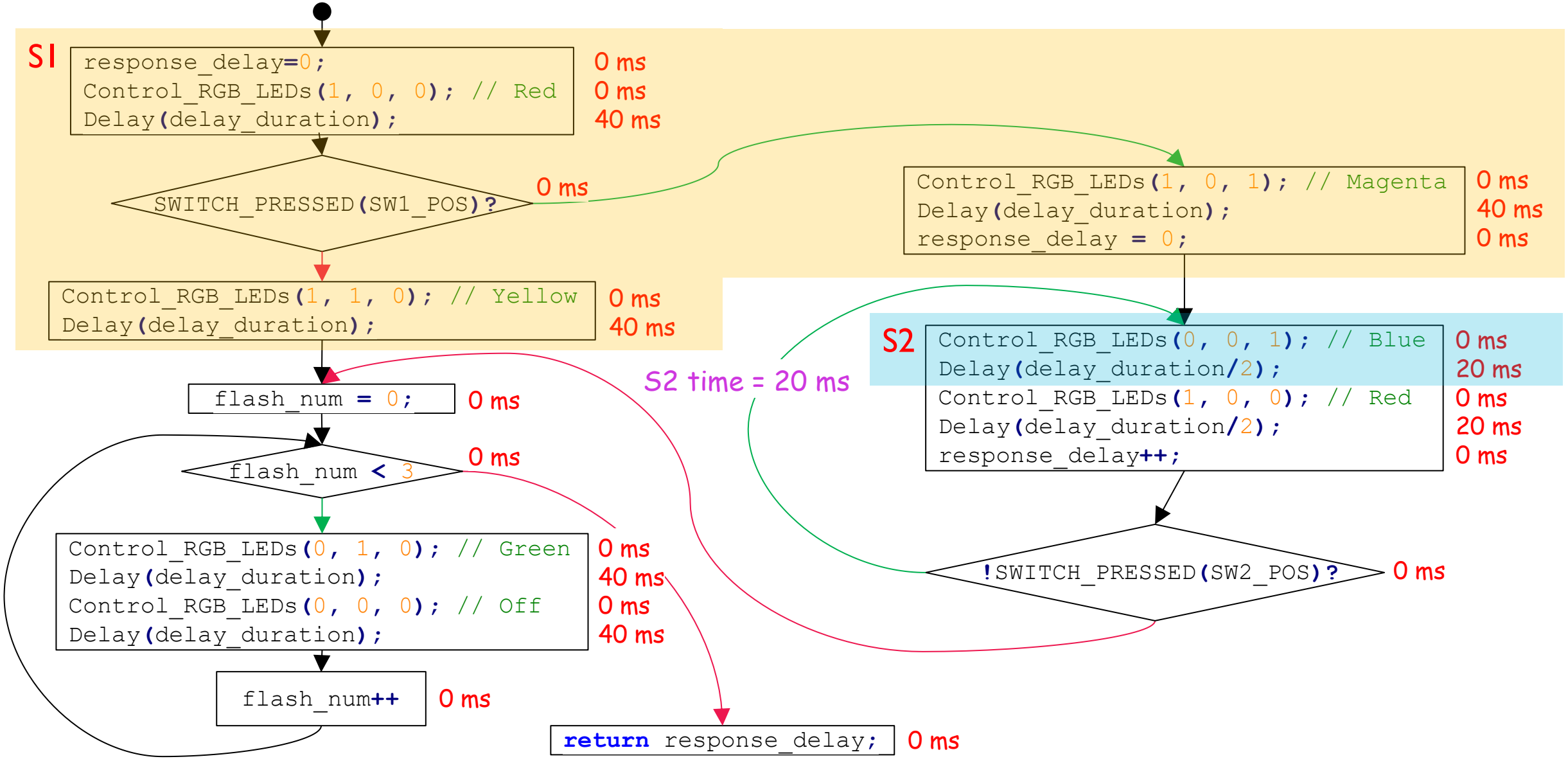
Final State S1 as Allowed by 250 ms Time Budget



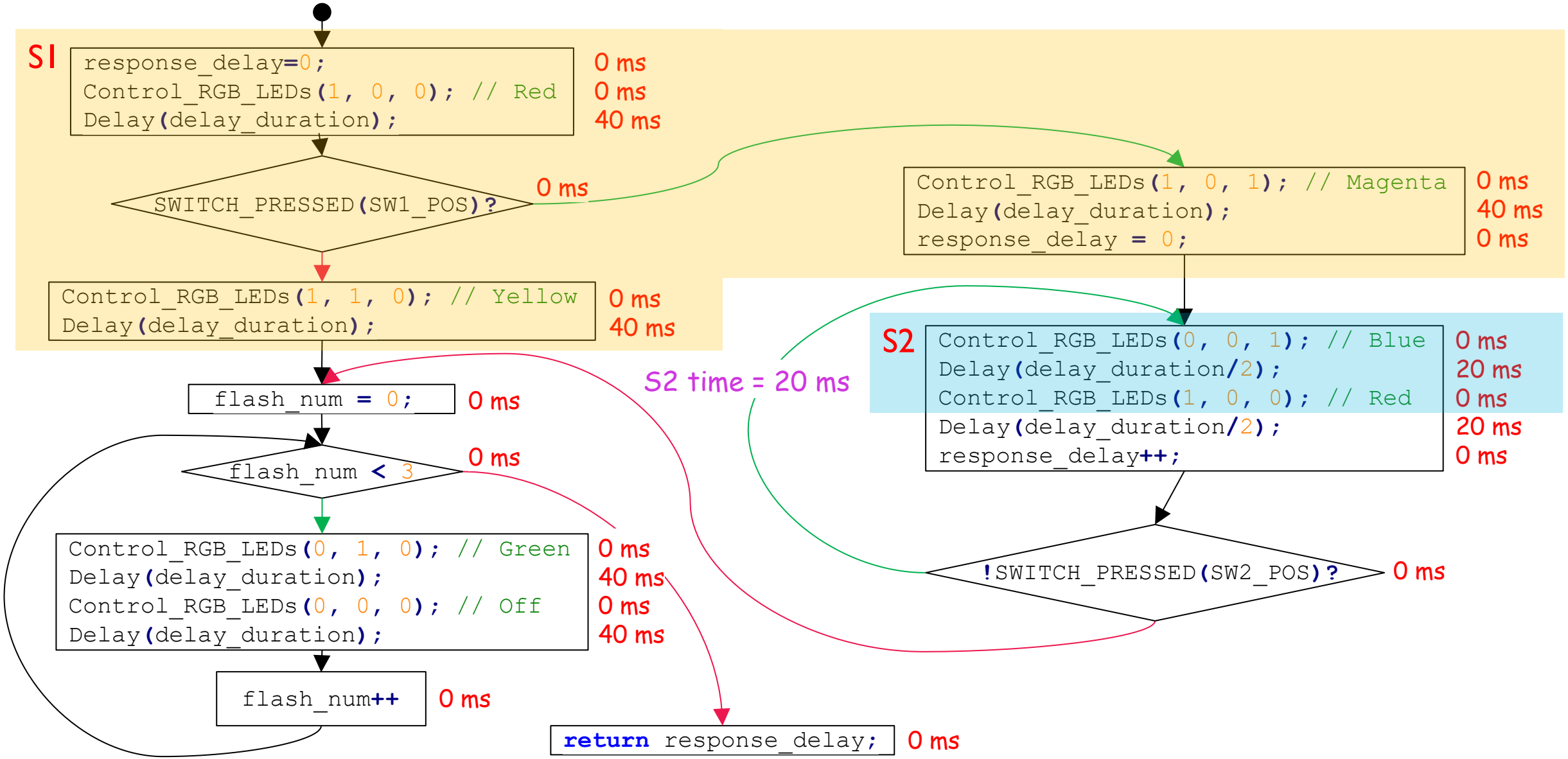
Add Code to State S2 as Allowed by 250 ms Time Budget



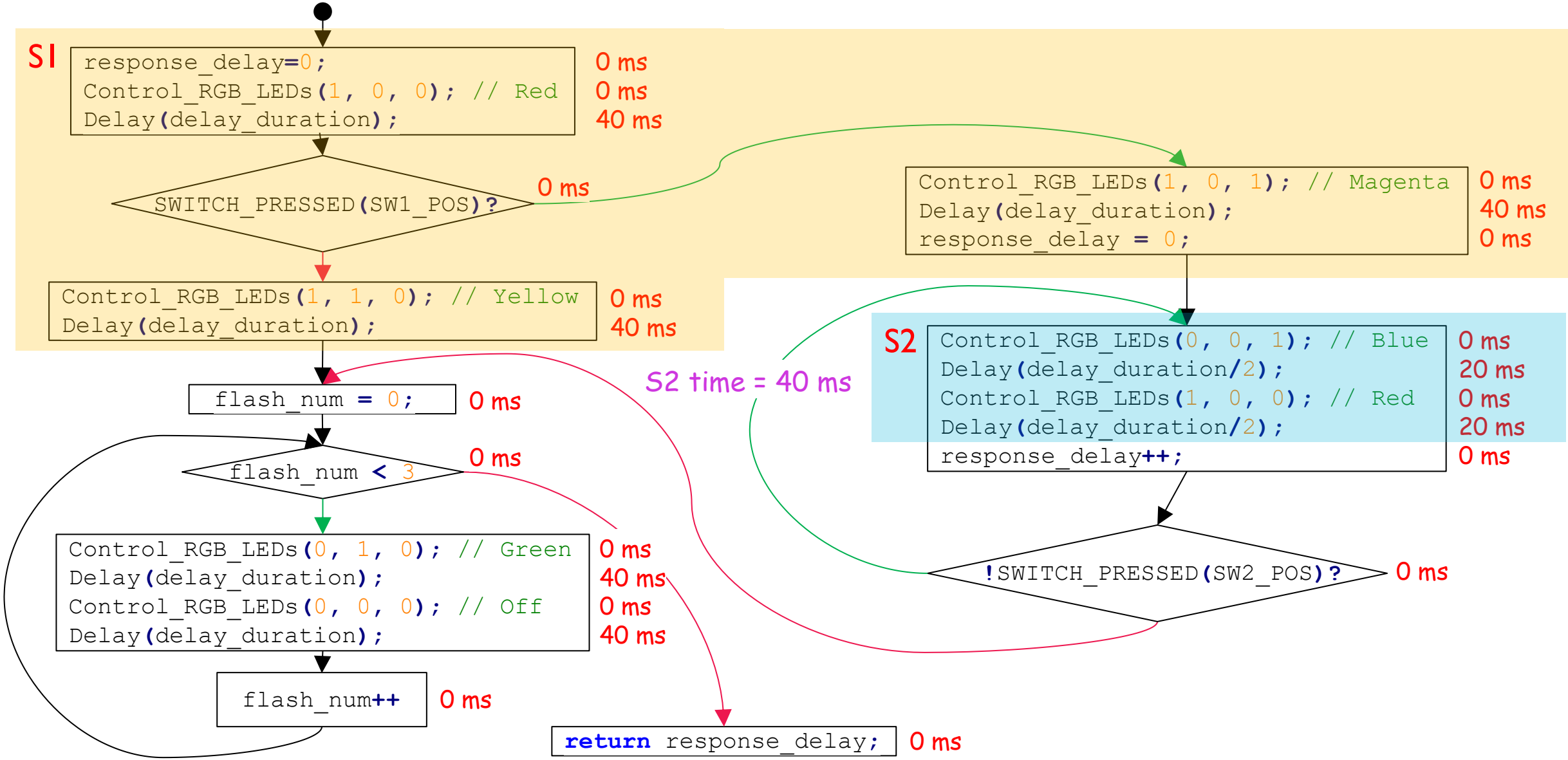
Add Code to State S2 as Allowed by 250 ms Time Budget



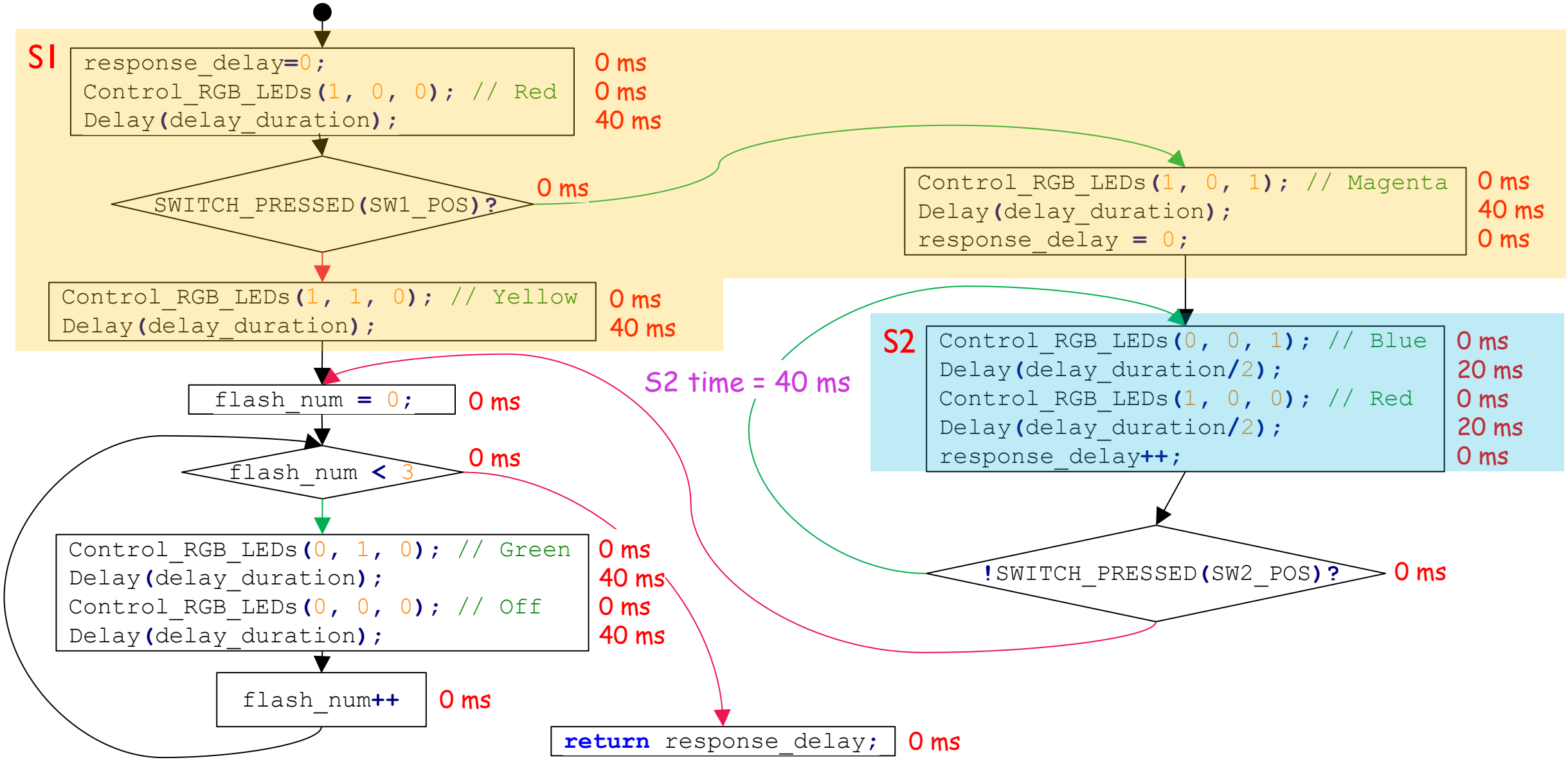
Add Code to State S2 as Allowed by 250 ms Time Budget



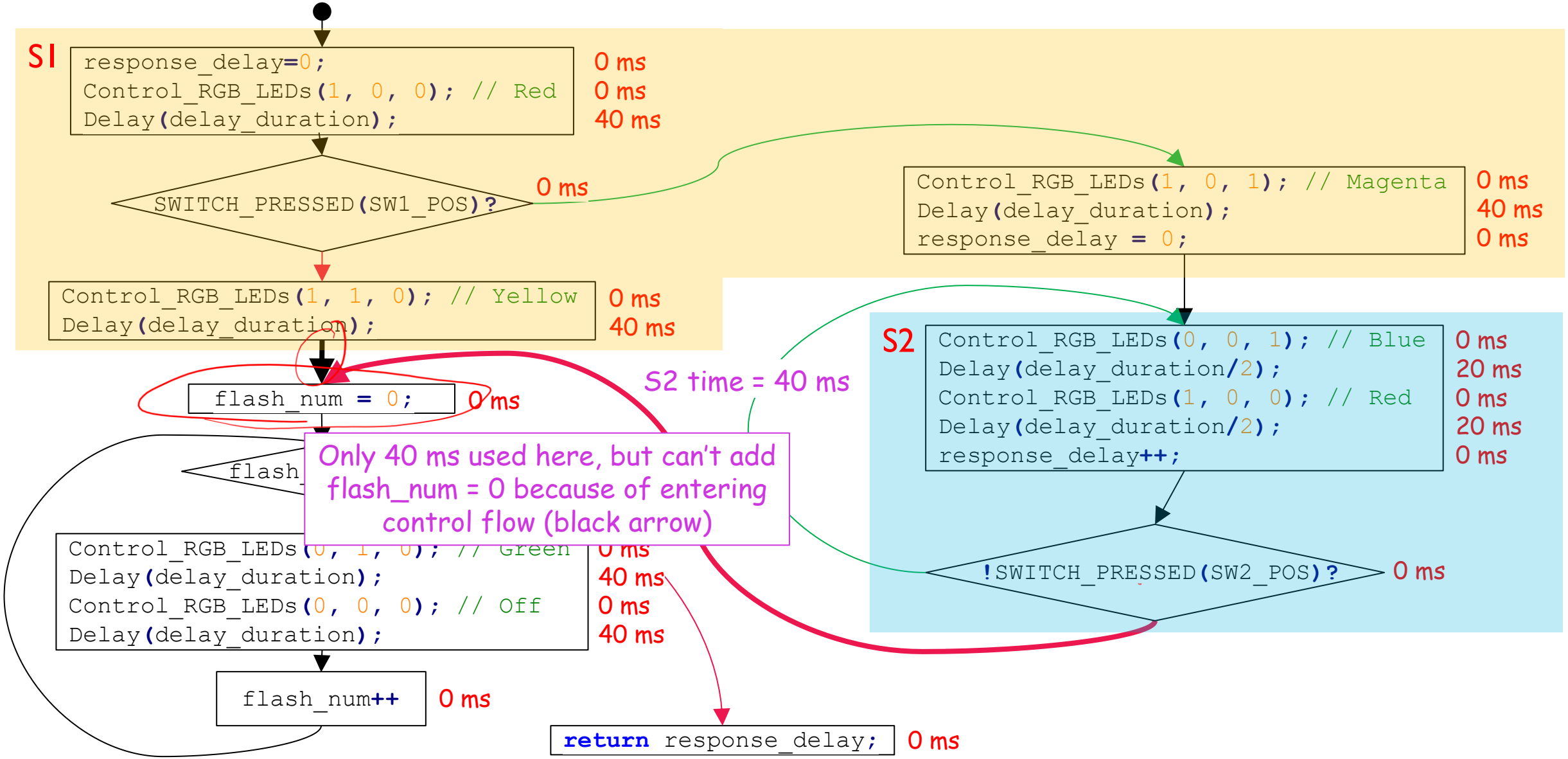
Add Code to State S2 as Allowed by 250 ms Time Budget



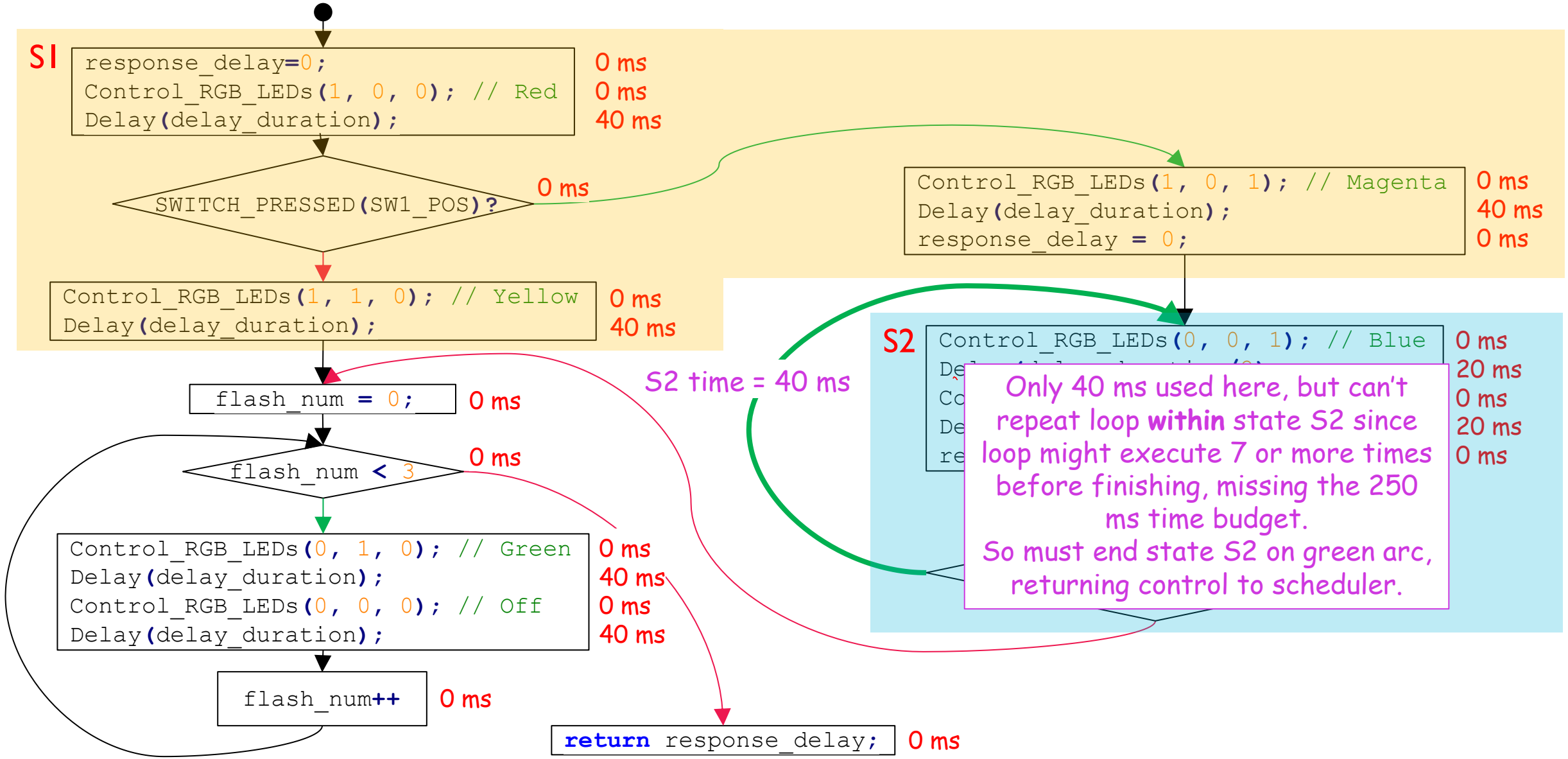
Add Code to State S2 as Allowed by 250 ms Time Budget



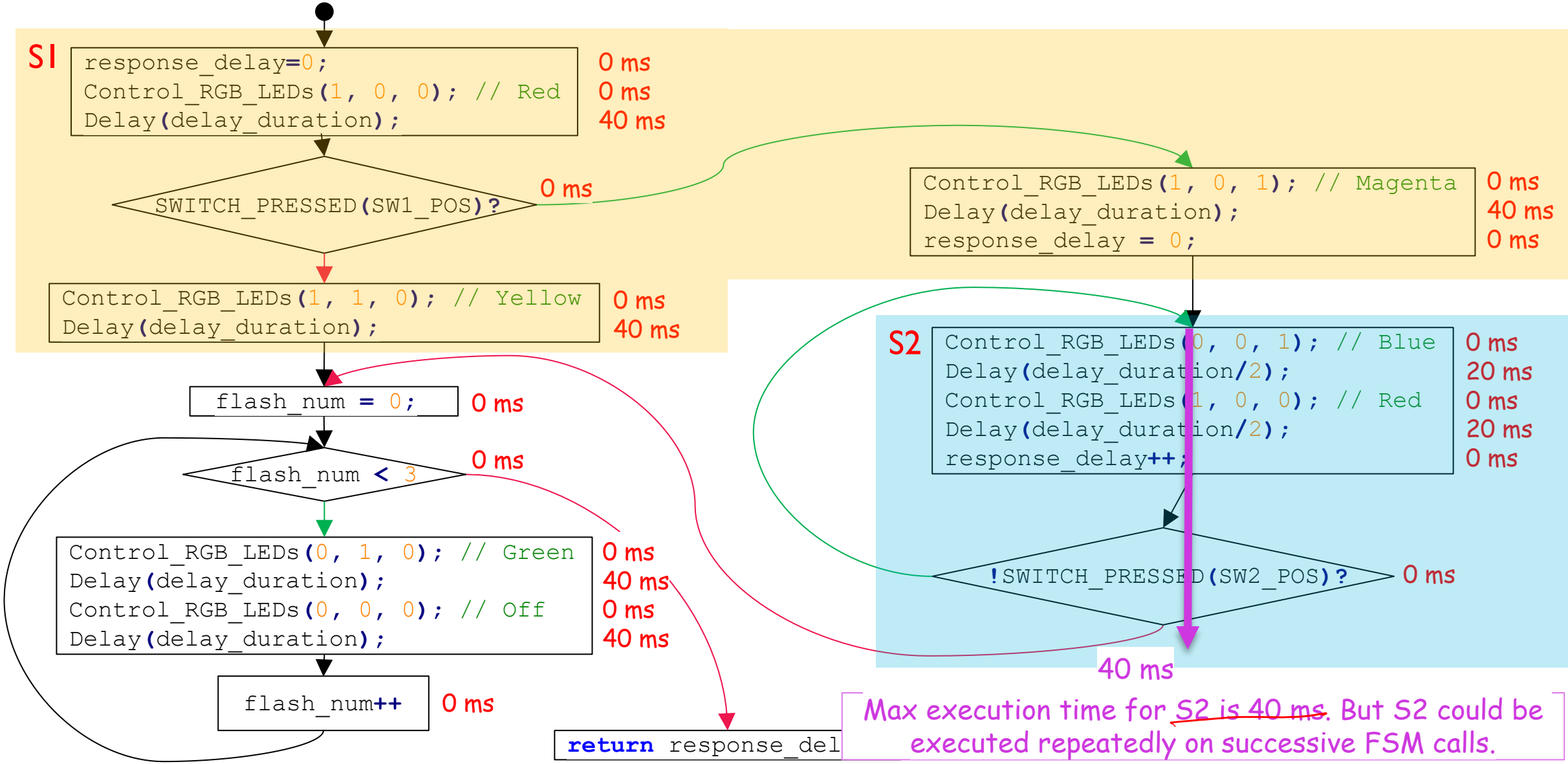
Add Code to State S2 as Allowed by 250 ms Time Budget



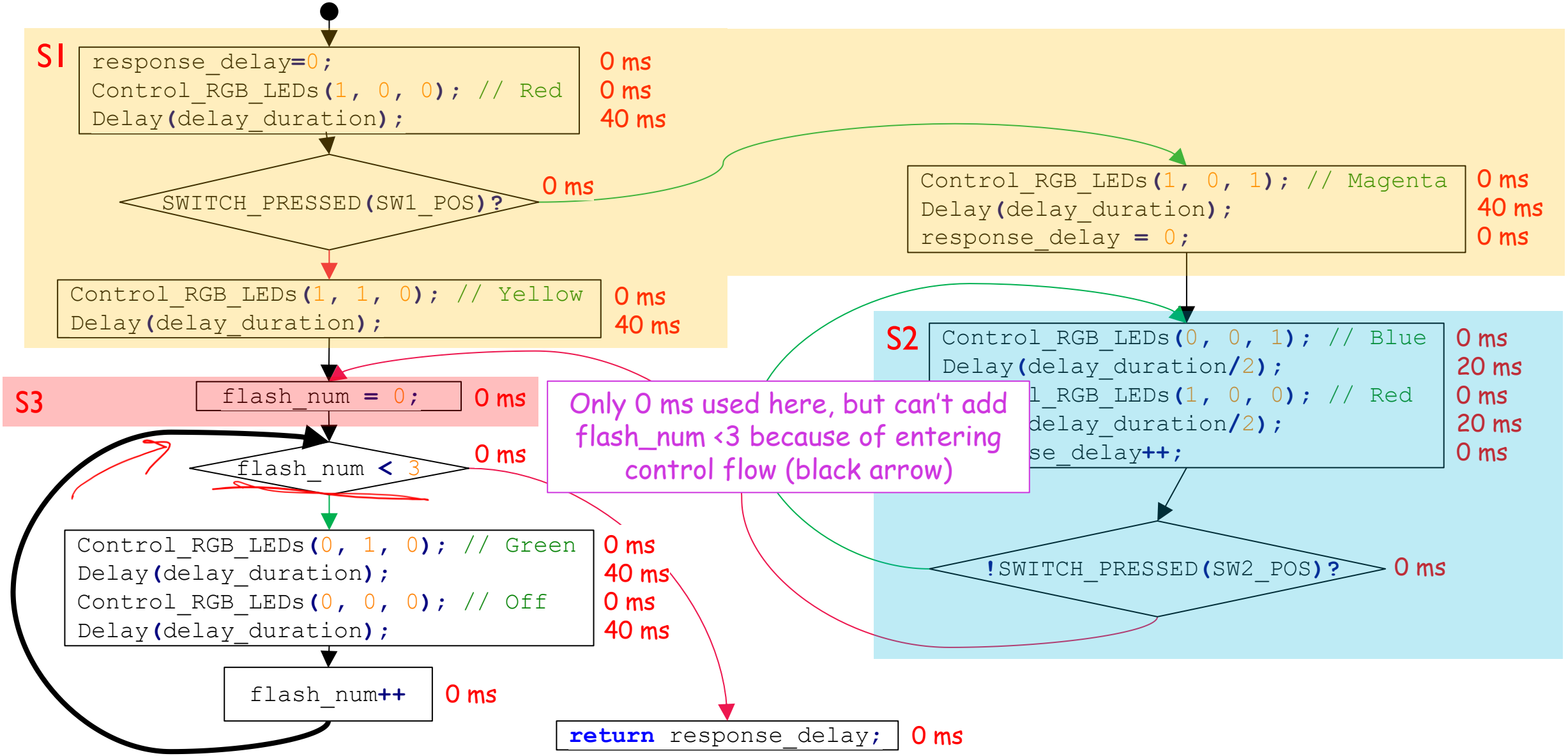
Add Code to State S2 as Allowed by 250 ms Time Budget



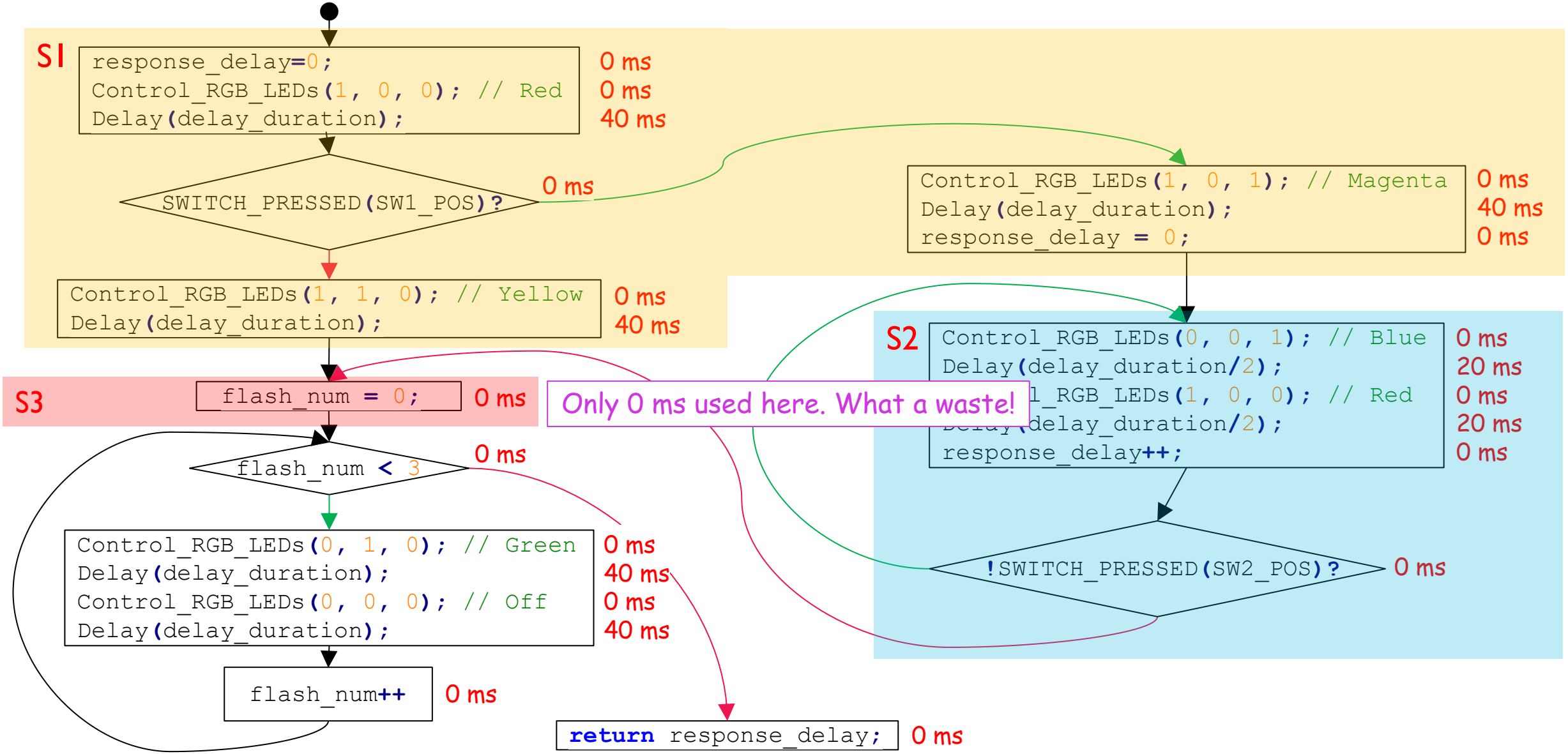
Final State S2 as Allowed by 250 ms Time Budget



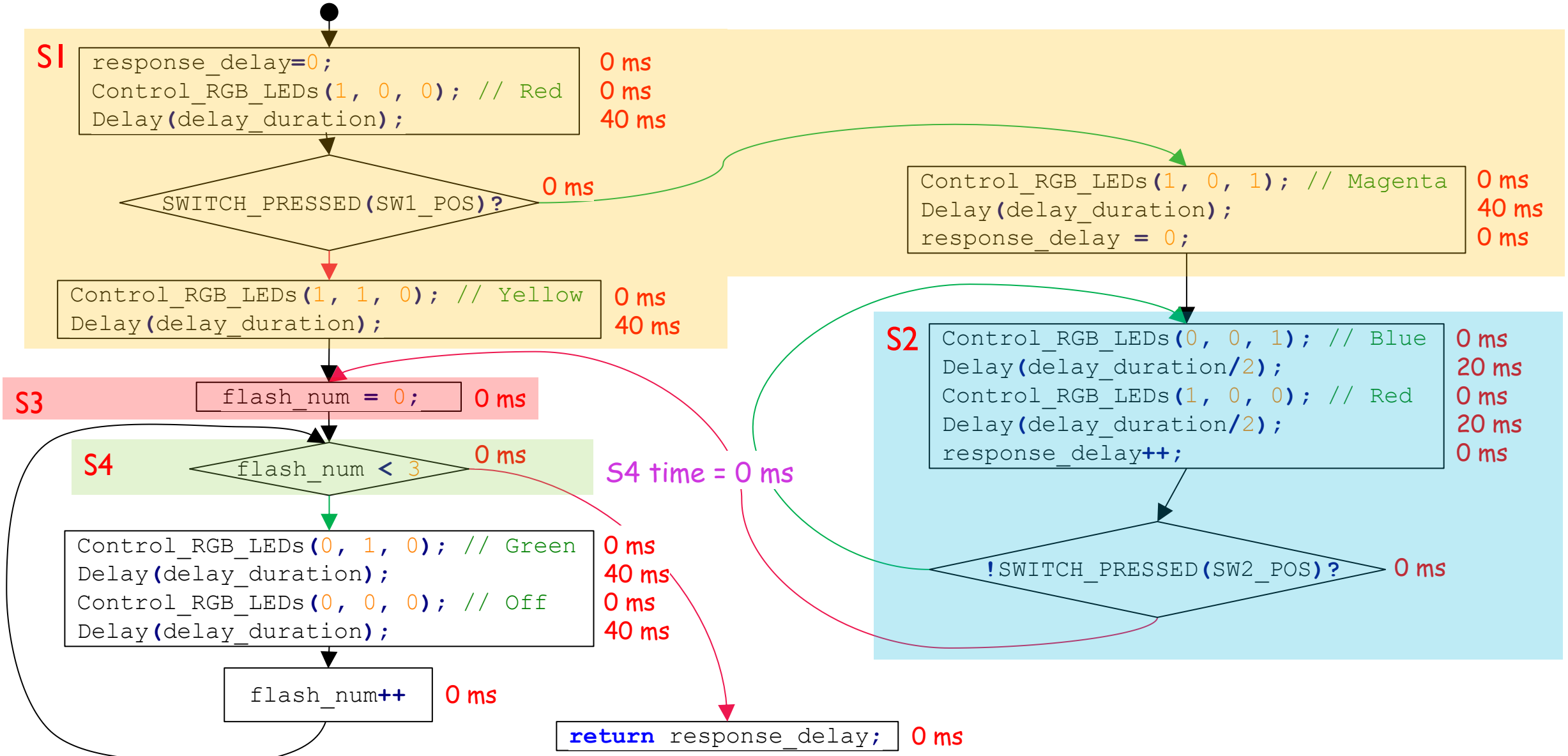
Add Code to State S3 as Allowed by 250 ms Time Budget



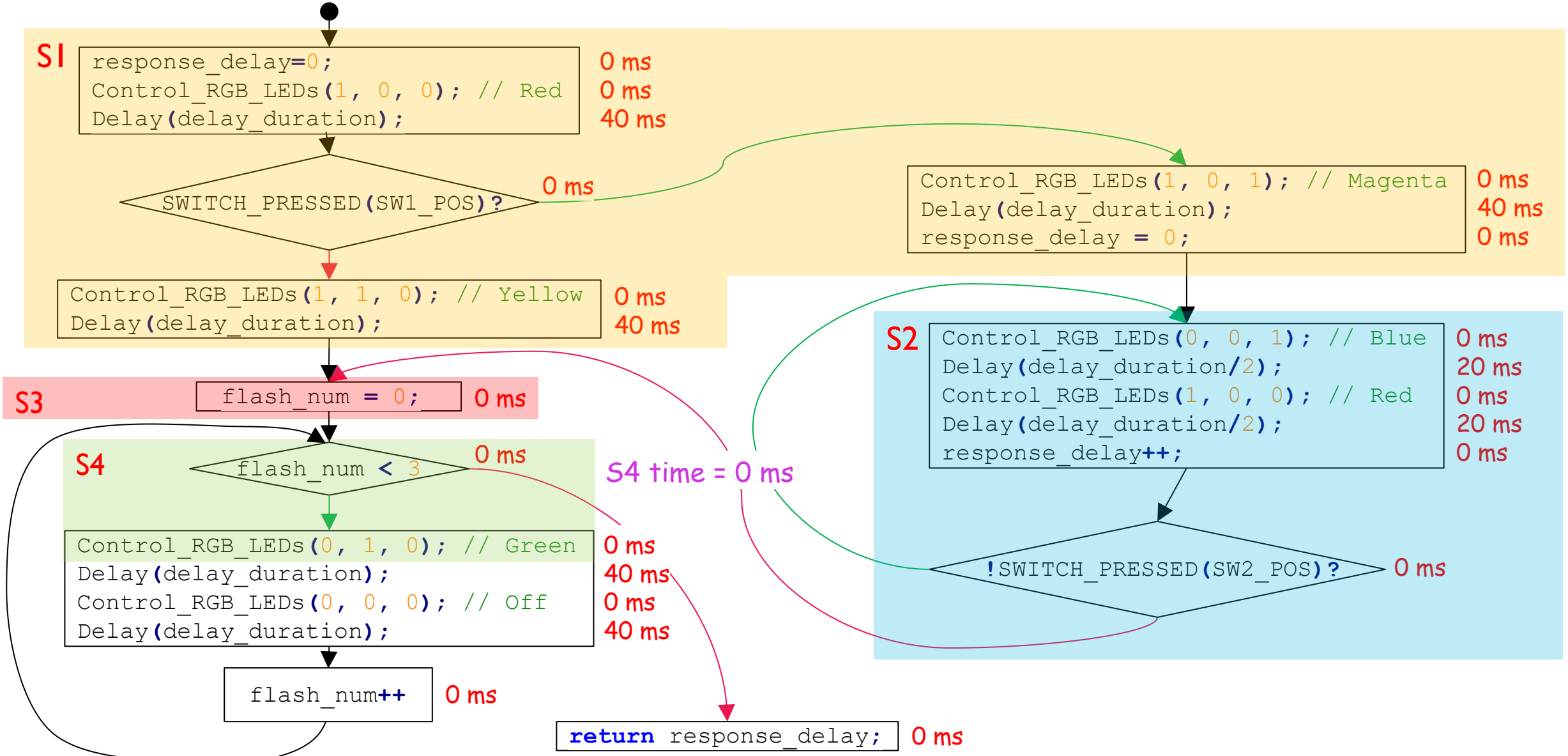
Final State S3 as Allowed by 250 ms Time Budget



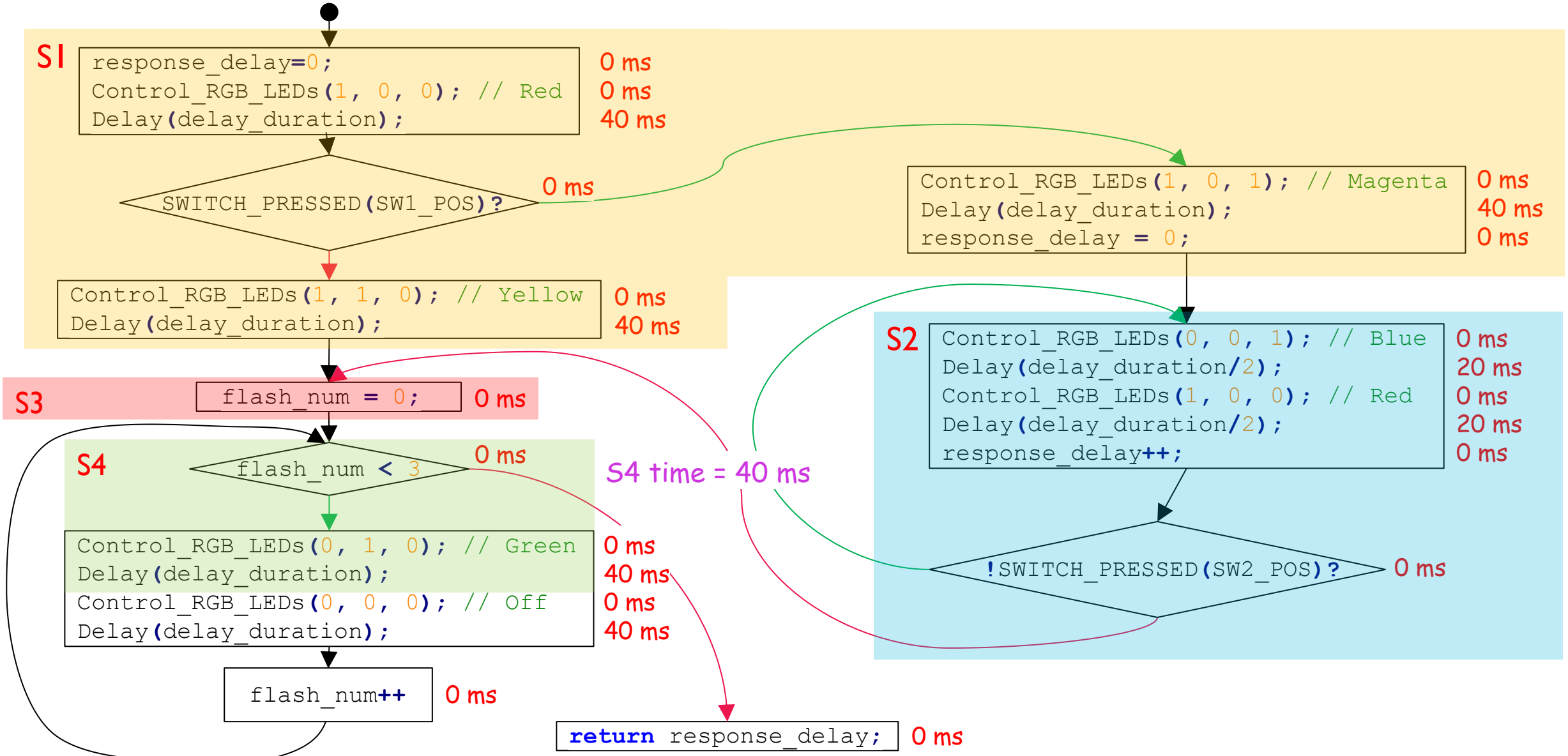
Add Code to State S4 as Allowed by 250 ms Time Budget



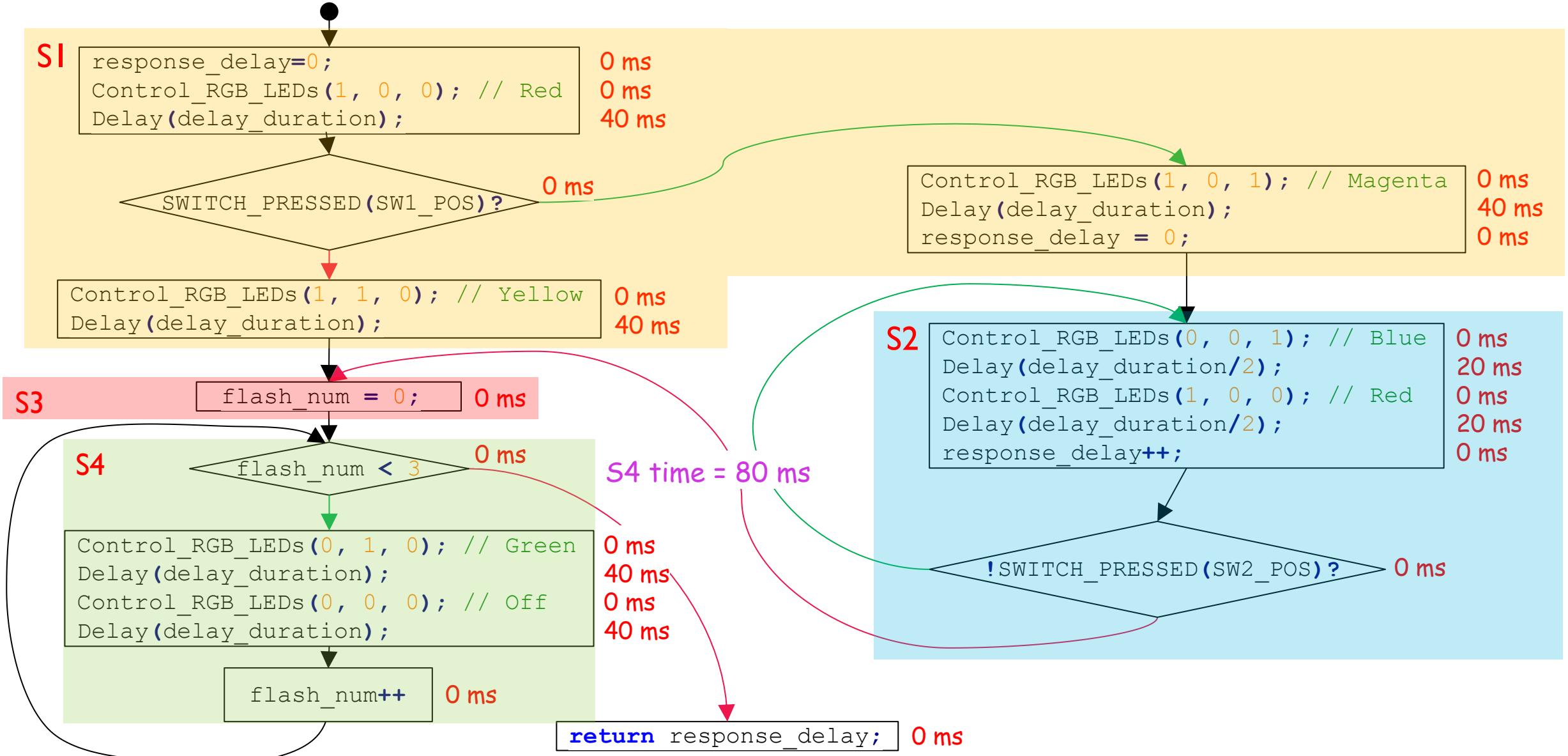
Add Code to State S4 as Allowed by 250 ms Time Budget



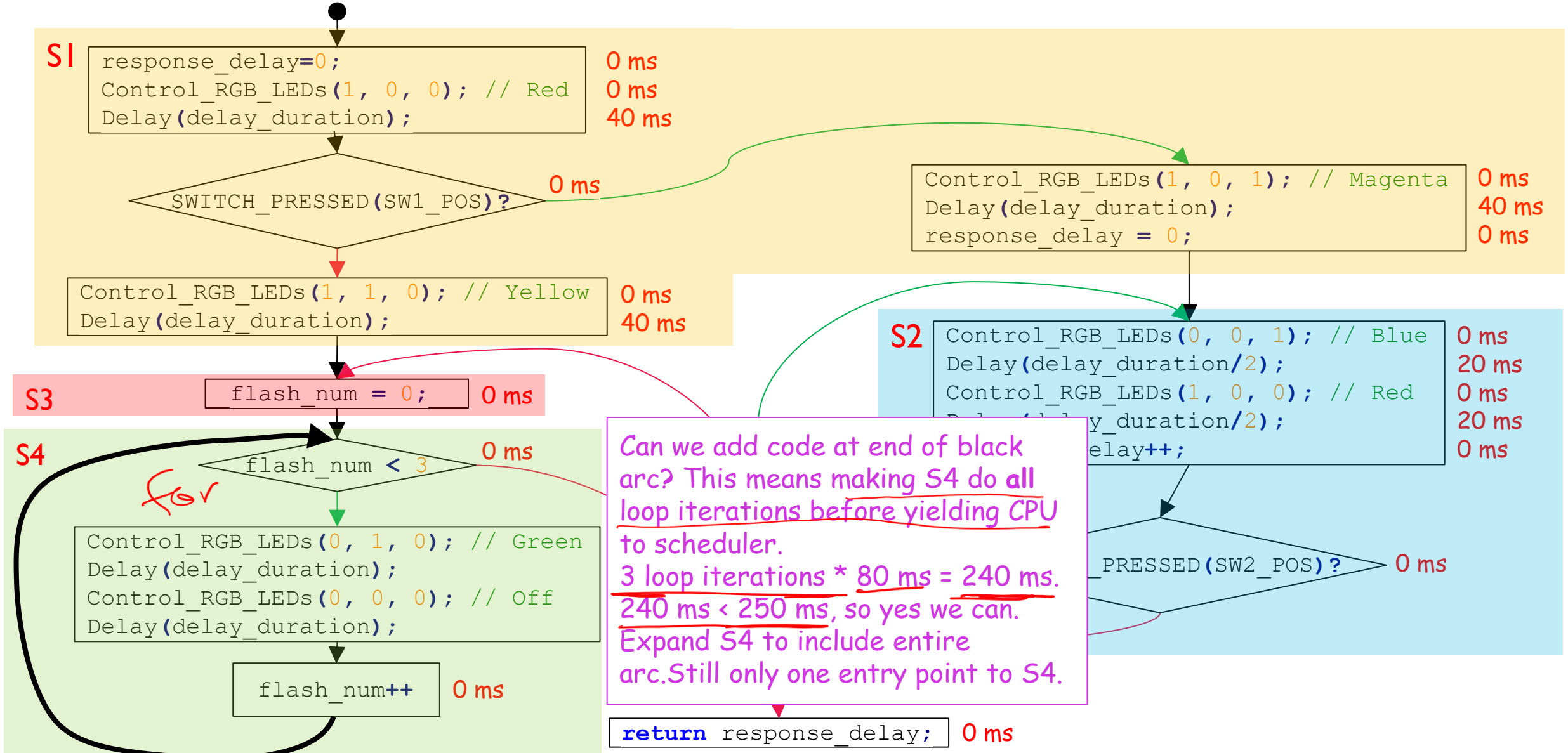
Add Code to State S4 as Allowed by 250 ms Time Budget



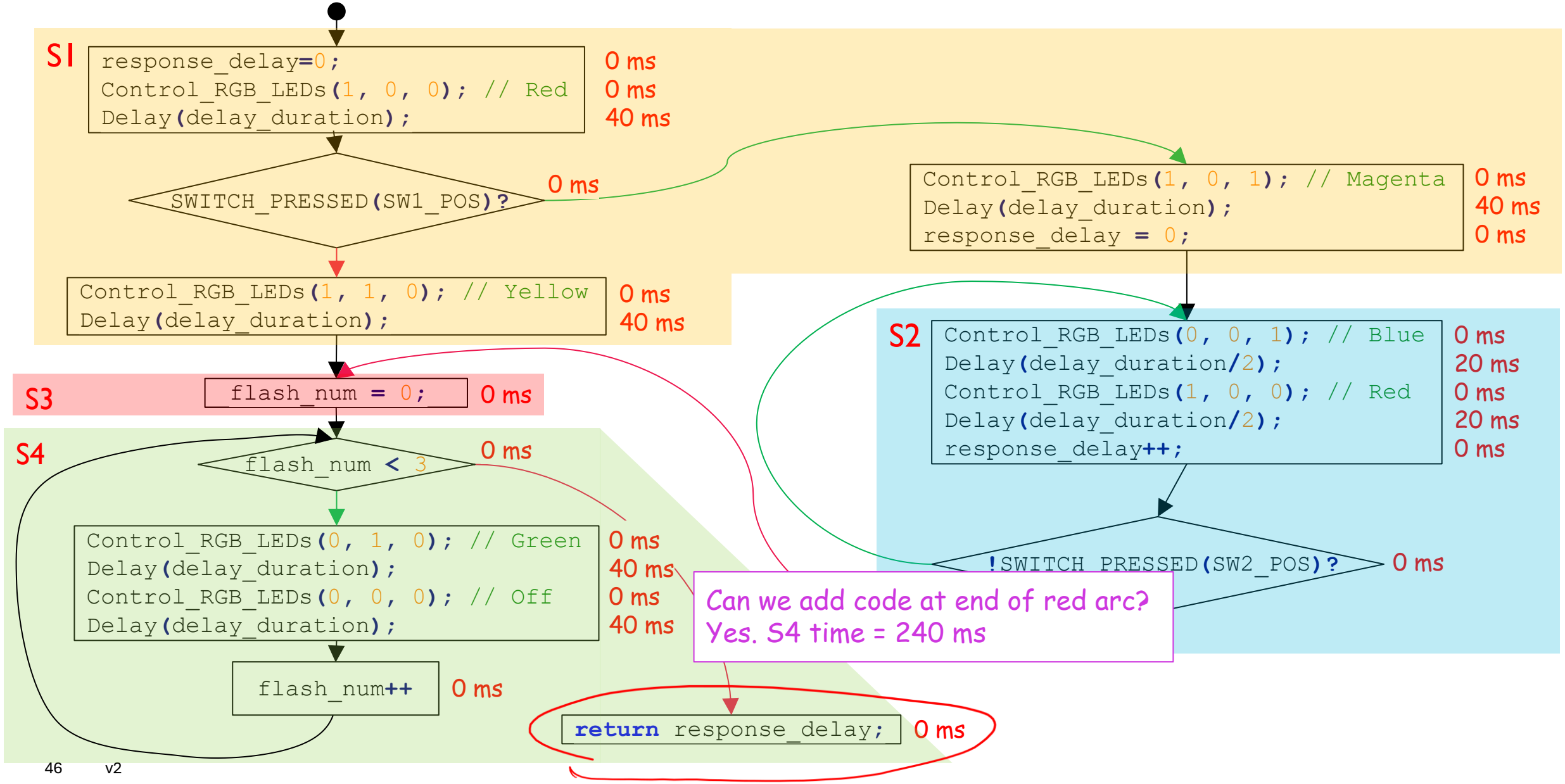
Add Code to State S4 as Allowed by 250 ms Time Budget



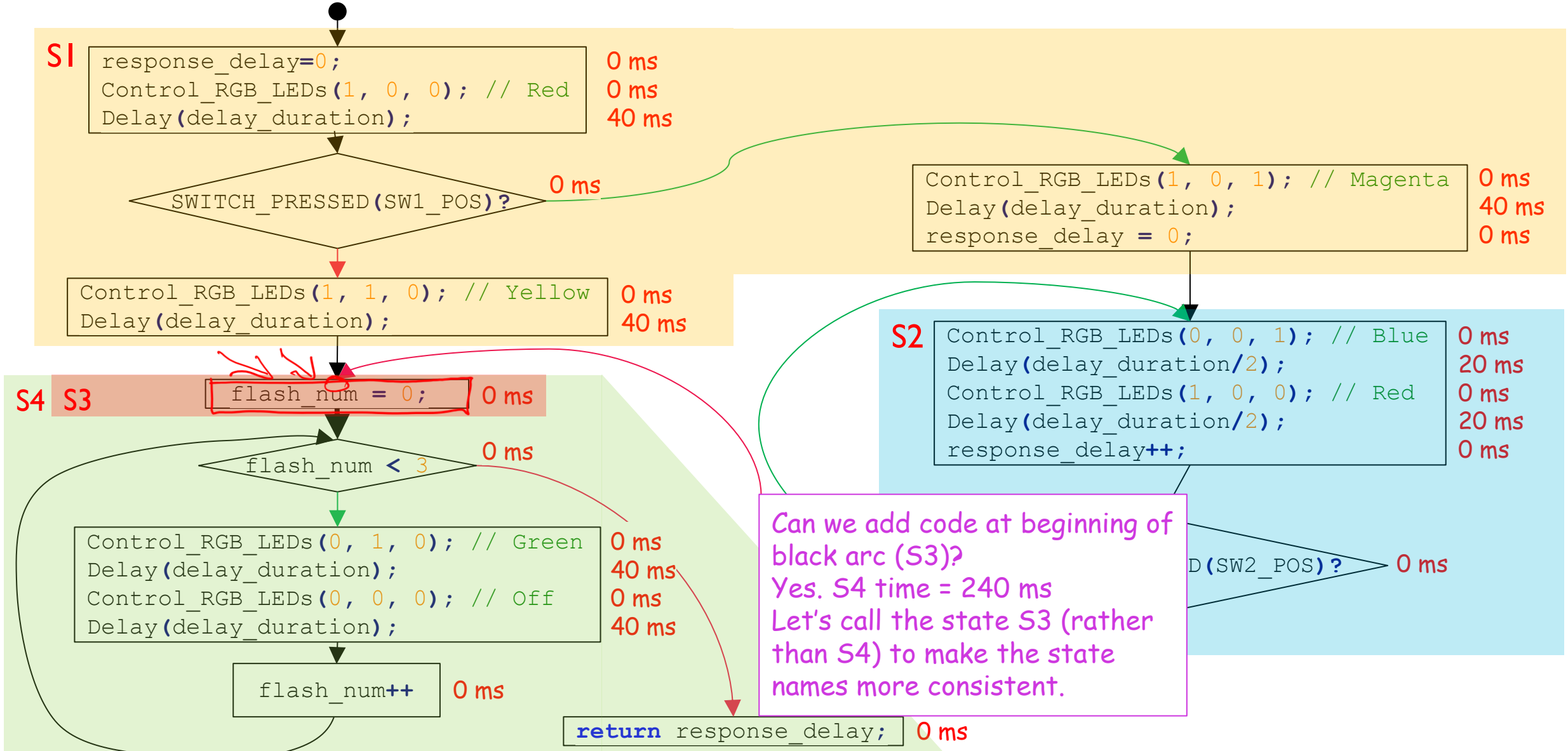
Add Code to State S4 as Allowed by 250 ms Time Budget



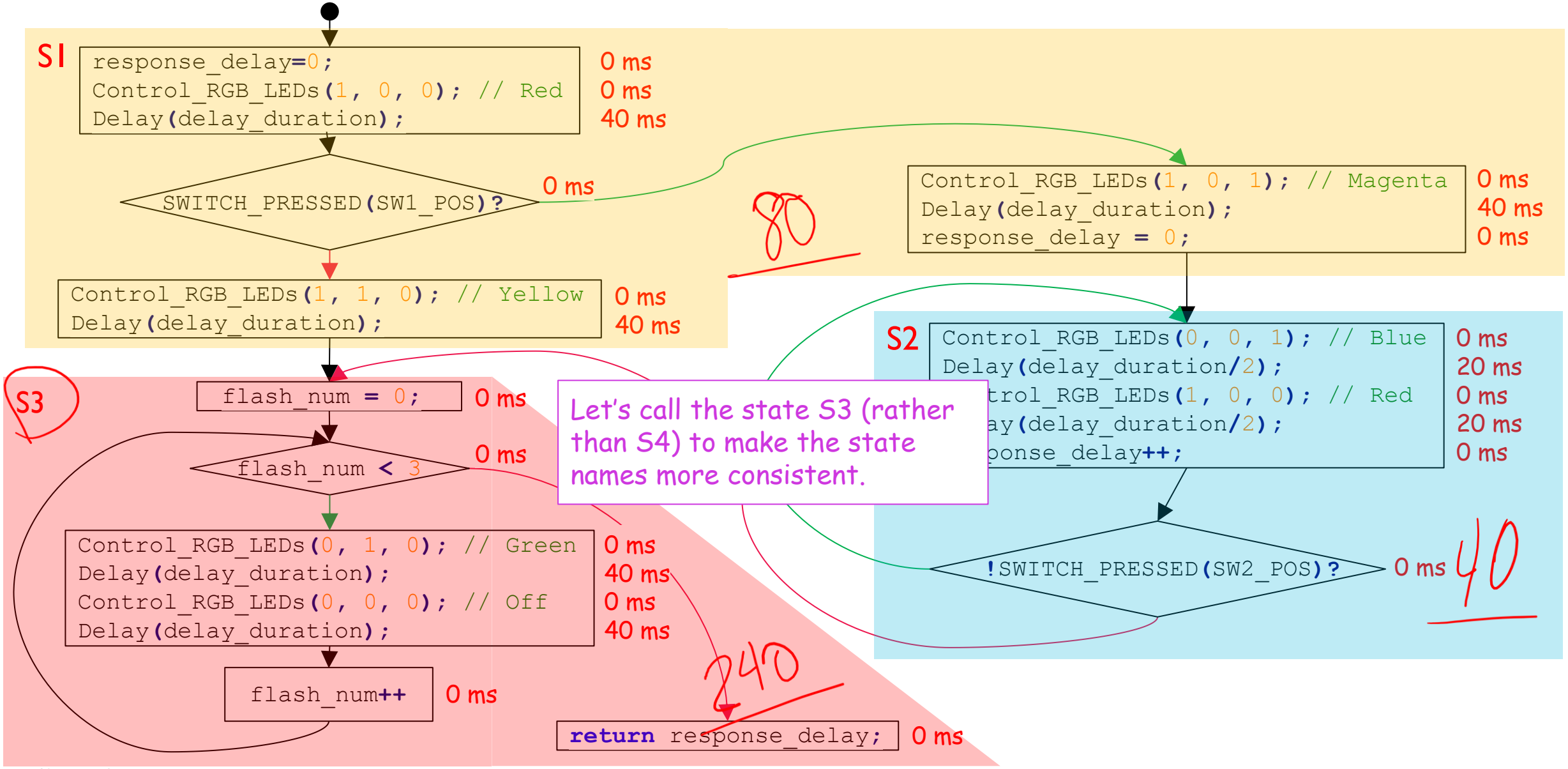
Add Code to State S4 as Allowed by 250 ms Time Budget



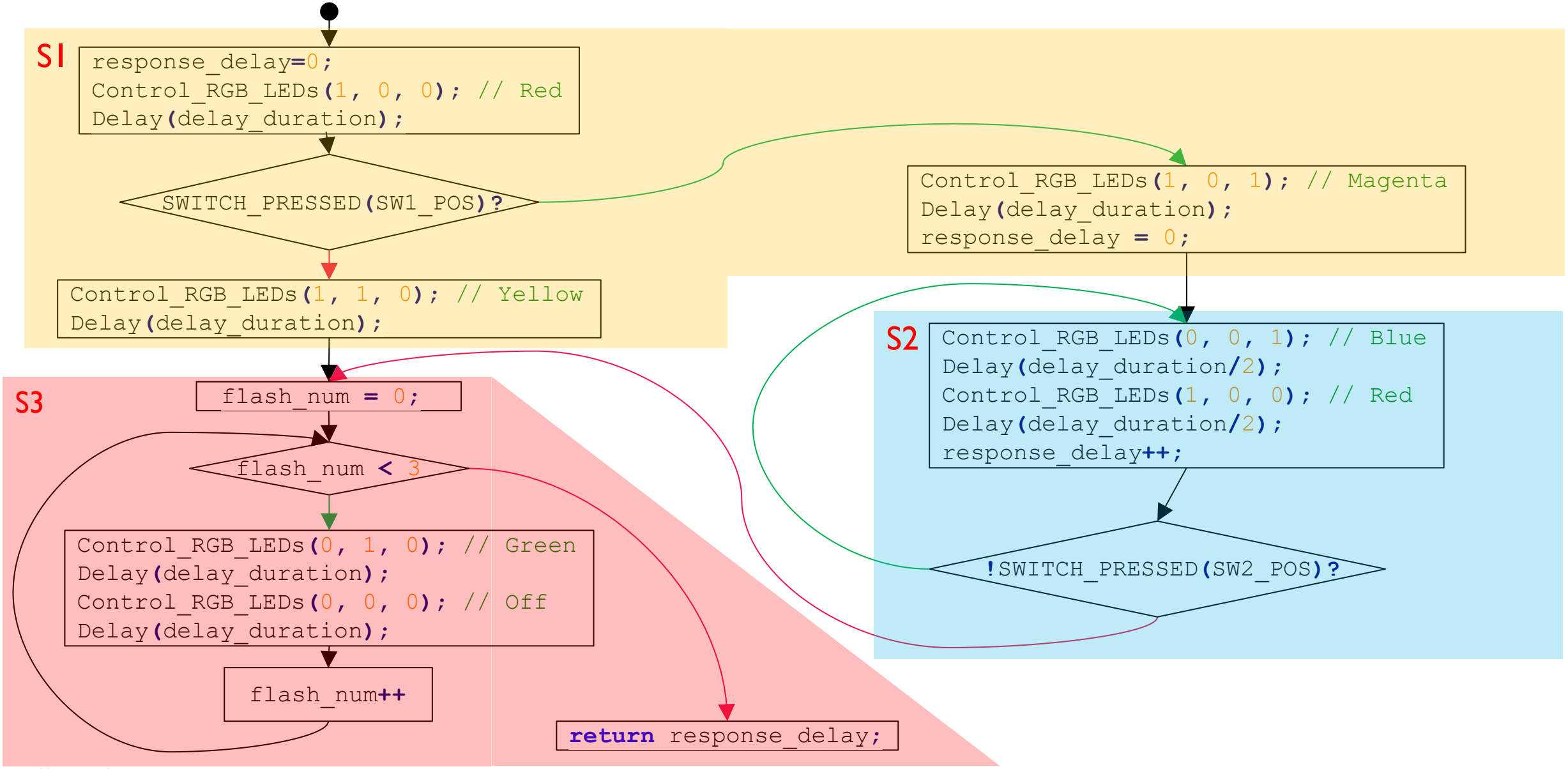
Add Code to State S4 as Allowed by 250 ms Time Budget



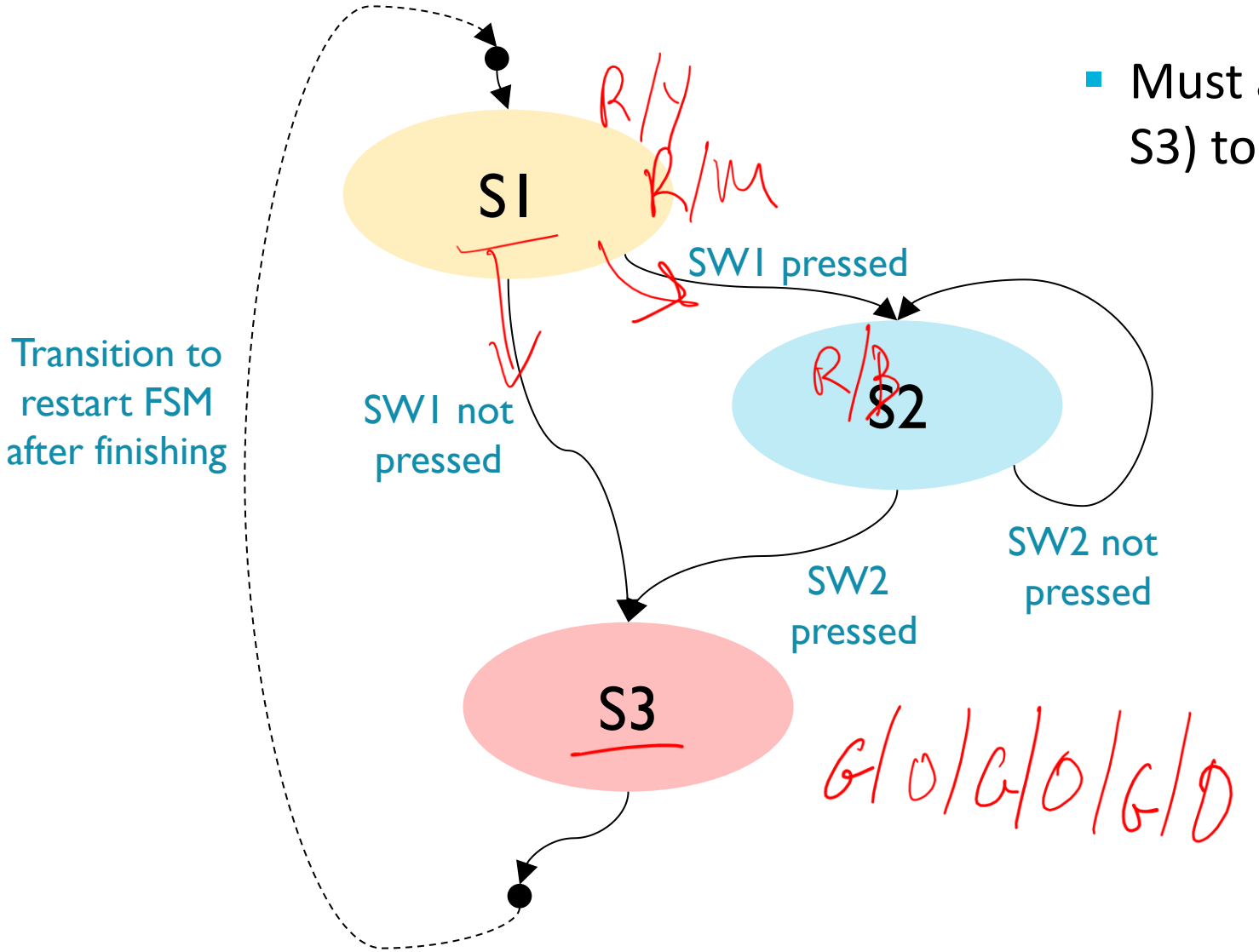
Final State S3 as Allowed by 250 ms Time Budget



CFG Marked with States



FSM Diagram

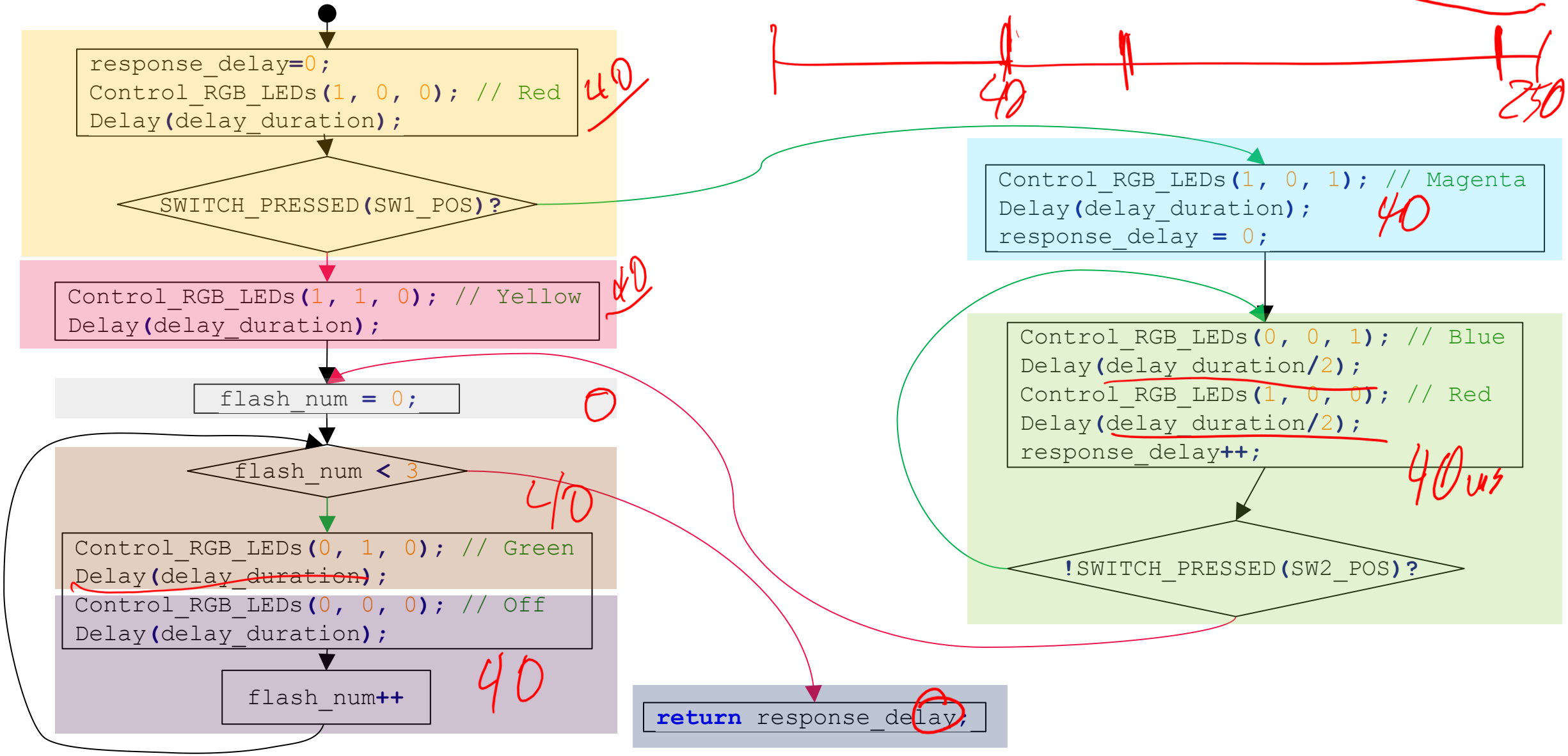


- Must add transition from exit (after S3) to entry (before S1)

Code for Example FSM (Incomplete)

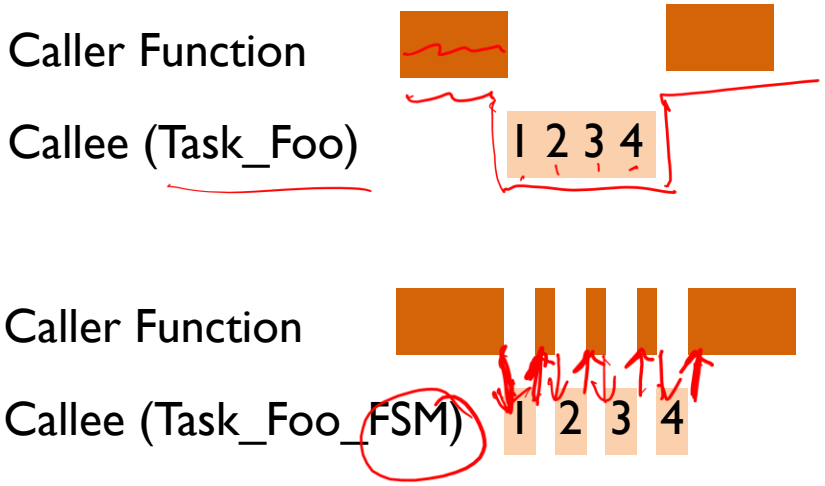
```
int Task_Color_Sequence_FSM(unsigned int delay_duration)
{
     int flash_num;
     static enum {S1,S2,S3} next_state = S1;
     static int response_delay=0, return_value=0;

     switch (next_state) {
          case S1:
               Control_RGB_LEDs(1, 0, 0); // Red
               Delay(delay_duration);
               if (SWITCH_PRESSED(SW1_POS)) {
                    Control_RGB_LEDs(1, 0, 1); // Magenta
                    Delay(delay_duration);
                    response_delay = 0;
                    next_state = S2;
               } else {
                    Control_RGB_LEDs(1, 1, 0); // Yellow
                    Delay(delay_duration);
                    next_state = S3;
               }
               break;
          case S2:
               Control_RGB_LEDs(0, 0, 1); // Blue
               Delay(delay_duration/2);
               Control_RGB_LEDs(1, 0, 0); // Red
               Delay(delay_duration/2);
               response_delay++;
               if (SWITCH_PRESSED(SW2_POS))
                    next_state = S3;
               else
                    next_state = S2;
               break;
          case S3:
               for (flash_num = 0; flash_num < 3;
               flash_num++) {
                    Control_RGB_LEDs(0, 1, 0); // Green
                    Delay(delay_duration);
                    Control_RGB_LEDs(0, 0, 0); // Off
                    Delay(delay_duration);
               }
               return_value = response_delay;
               next_state = S1;
               break;
          default: next_state = S1;
               break;
     }
     return return_value; // Not always valid!
}
```

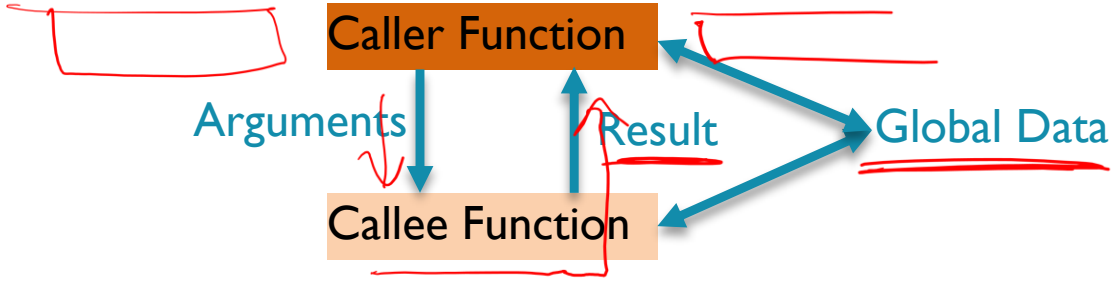
Many Other Possible FSMs for the Function: One Example



Synchronization and Communication

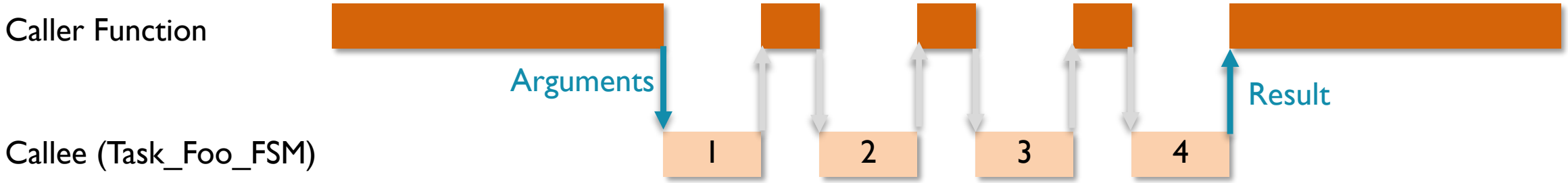
Understanding the Problem





- Implications of turning a run-to-completion function into FSM-based task
  - May need to call function Task_Foo_FSM many times to get all the work done
  - We have de-synchronized the timing relationship between caller function and callee
  - Must change how caller and callee communicate data
- Data communication for callee function
  - Arguments: input values
  - Result: return value(s)
  - Global data: can be inputs, outputs or both. Can be variables,data structures, etc.

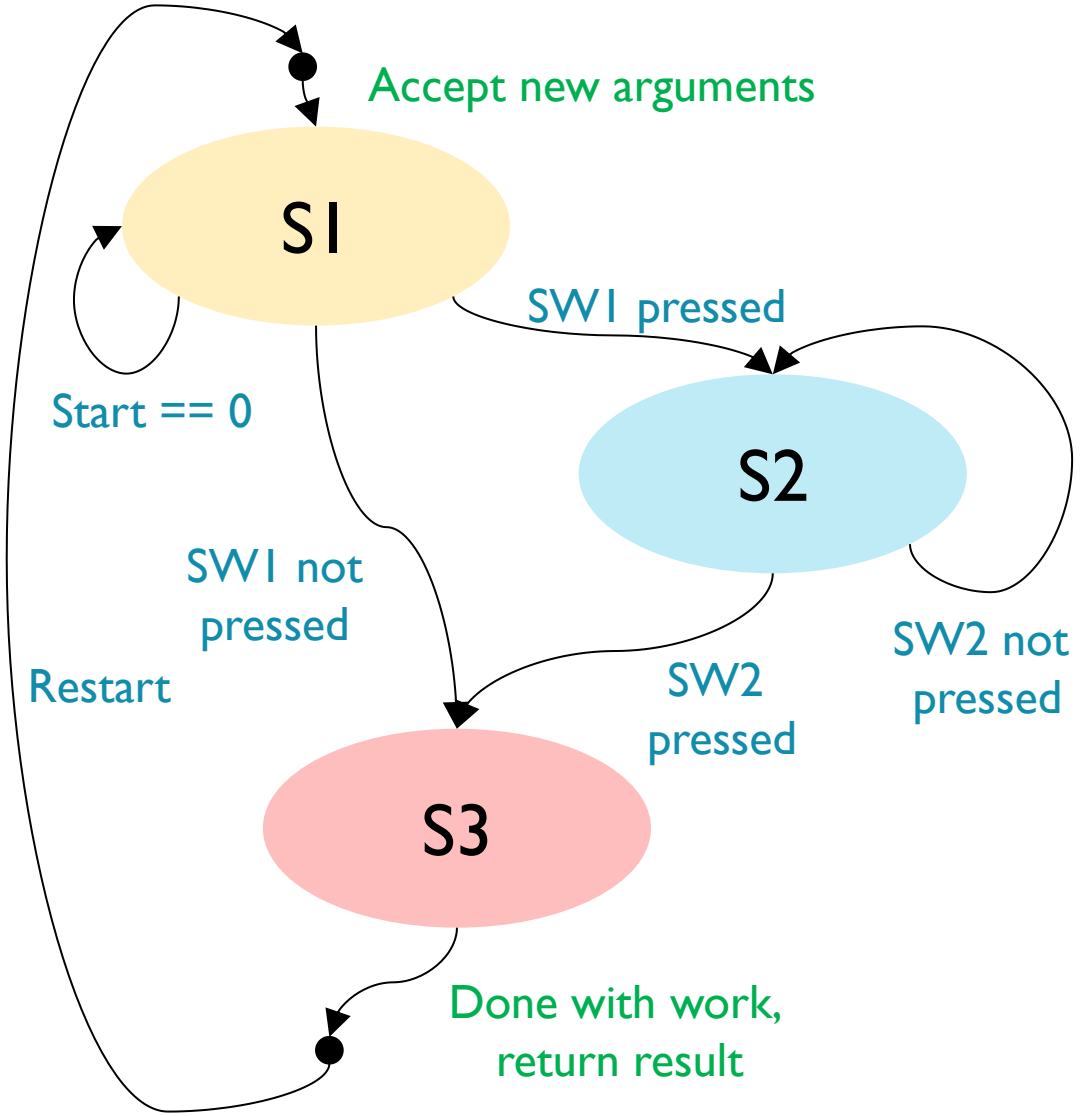
Data Communication Problems



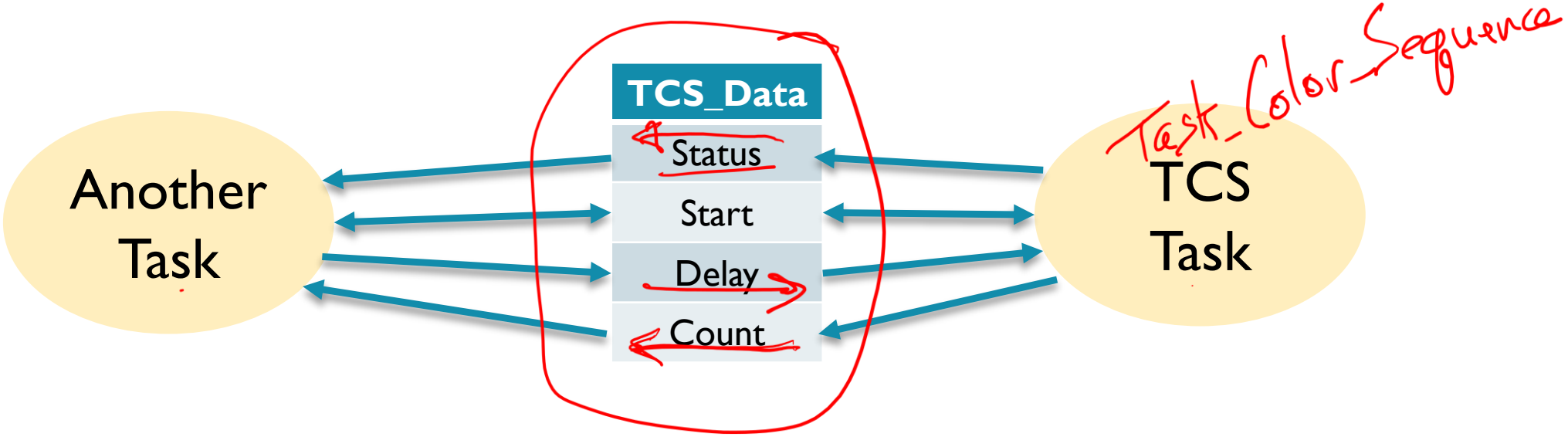
- When can we pass new arguments?
  - Task_Foo: each time it is called
  - Task_Foo_FSM: only if next_state = first state
- When can we use the return value?
  - Task_Foo: each time it returns
  - Task_Foo_FSM: only when FSM has completed a state which defines the return value
- What about using global variables?
  - Can be read or written at almost any time
  - Must prevent race conditions by completing each access before ending state
- The calling function is the scheduler, so we can simplify.
  - Do we really need arguments and results?
    - Probably not. We'd have to build application logic into the scheduler to do that
  - Rearchitect application to share data in a different way with global data objects

Synchronization and Data Exchange

- Caller function and FSM function need to synchronize and exchange data
- Start
  - Sync: When is the FSM ready to accept new inputs (formerly function arguments)?
  - Data Exchange: Where do we put the input data?
- End
  - Sync: When has the FSM completed its work?
  - Data Exchange: Where do we put the output data?

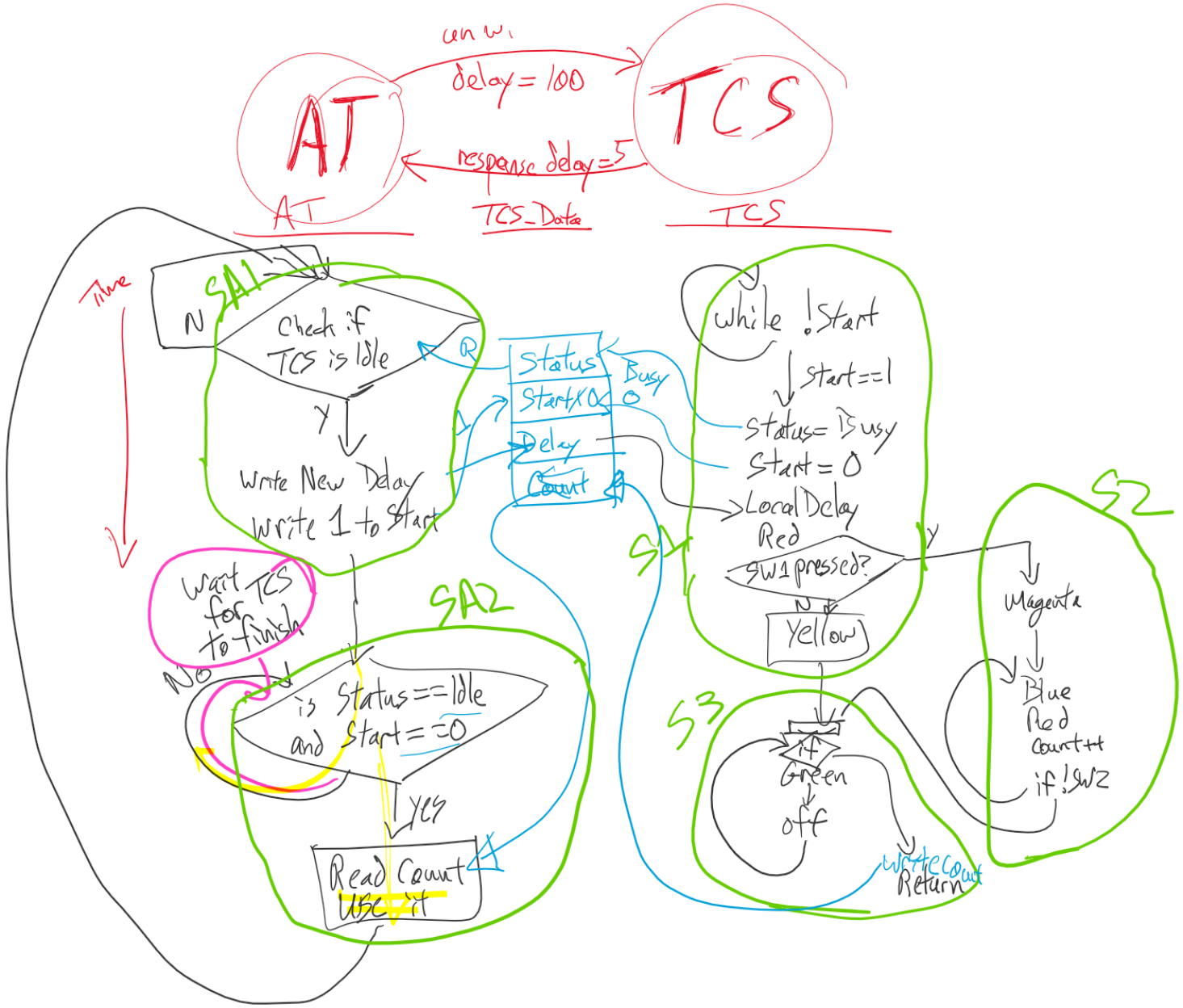


Data Structure with Task Information

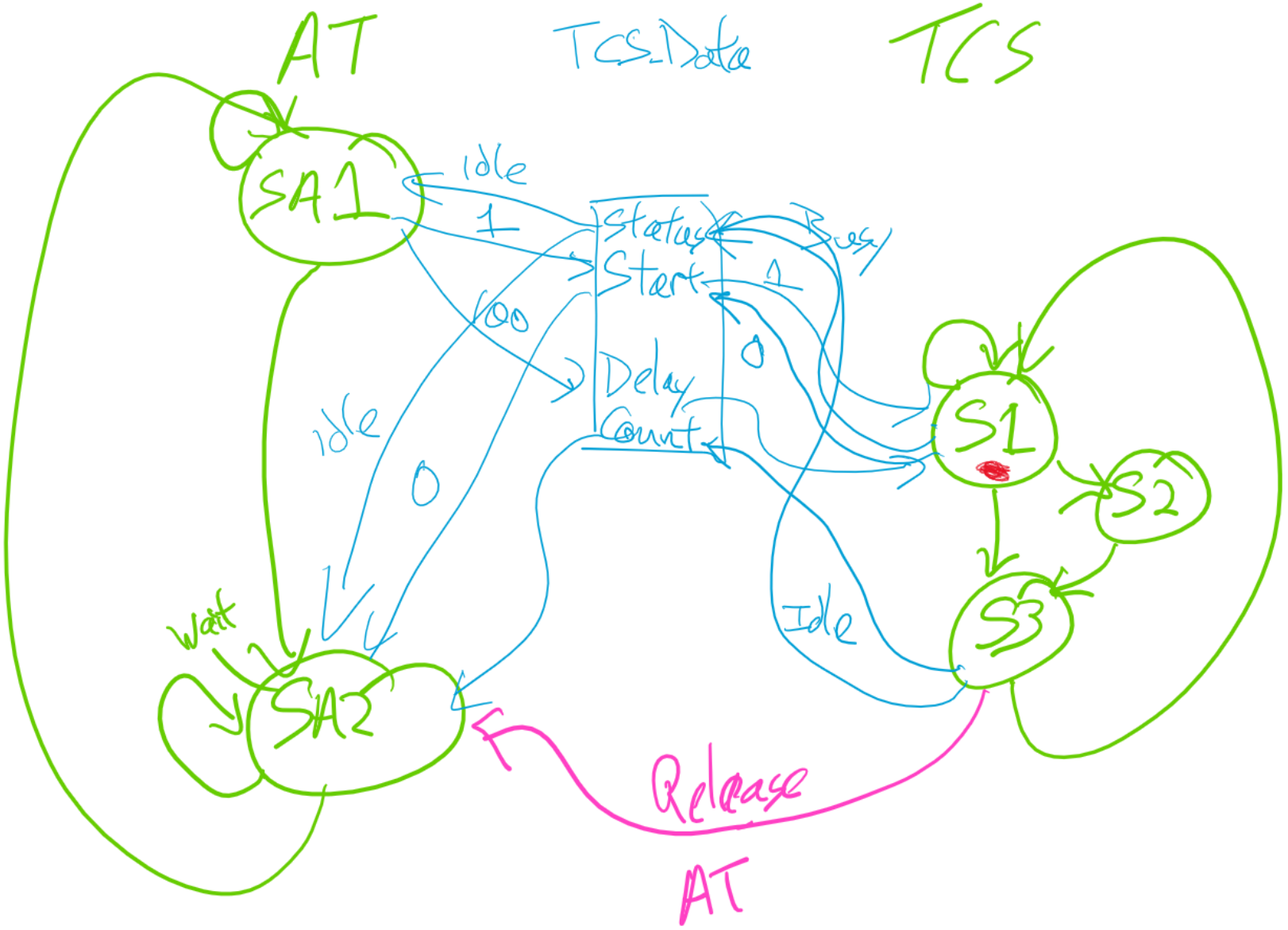



```
typedef struct {
  enum {
    FSM_IDLE = 0,    // FSM completed, ready for new data
    FSM_BUSY         // FSM busy
  } Status;
  uint16_t Start;
  uint32_t Delay;
  uint16_t Count;
} TCS_DATA_T;

TCS_DATA_T TCS_Data={FSM_IDLE, 0, 0, 0};
```

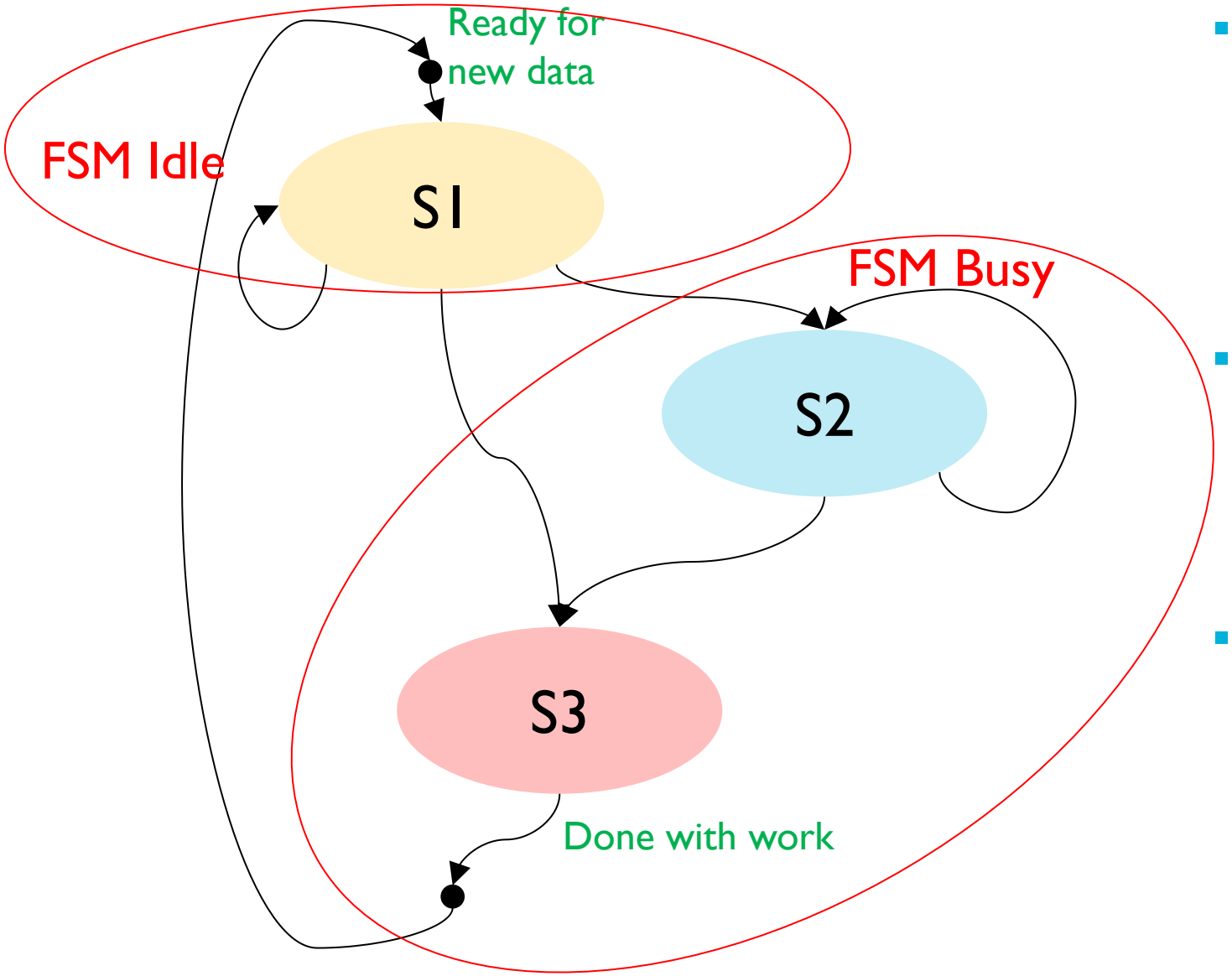
FSM Communication Example



Communication Example (Simplified)



FSM Status Definitions



- What is the status of the FSM?
  - FSM_IDLE: FSM is ready to accept new data and start processing it
  - FSM_BUSY: FSM still working
  - Could have additional status values (e.g. if accepting series of data items over time)
- Determine status from value of next_state after completing this state
  - S1? Done with current work, ready for new work, so FSM will be Idle
  - S2? S3? Still working, so FSM will be Busy
- Other tasks can use this information for handshaking (determining when they can send information)

Accepting New Inputs

- Use FSM status to determine when to provide and accept new data
- Example: delay period for LED flasher
  - Copy Delay only when in FSM_IDLE (S1)
- Code modifications
  - Declare static local variables to retain input value (delay_duration)
  - Add code to copy inputs to local variables in FSM_INIT

Updating and Reading FSM Status

- Use FSM status to determine when return value is valid and can be used
- Example: count number of times through red/blue loop, return that value
  - Uses Count field
- Code modifications
  - Type definition of structure to hold count and FSM status
  - Set FSM status to busy before exiting first (idle) state
  - Set FSM status to idle before exiting states which transition to idle state

Final Code for Example FSM

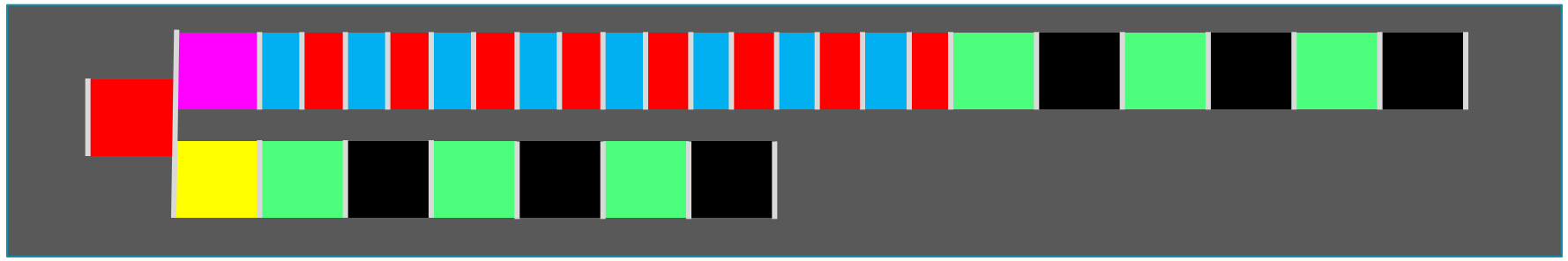
```
void Task_Color_Sequence_FSM(void) {
   int flash_num;
   static unsigned int delay_duration=100;
   static enum {S1,S2,S3} next_state = S1;
   static int response_delay=0;

   switch (next_state) {
      case S1:
         if (TCS_Data.Start > 0) {
            // Clear start request
            TCS_Data.Start = 0;
            // Accept new input data and start processing
            delay_duration = TCS_Data.Delay;
            // Initialization
            response_delay = 0;
            TCS_Data.Count = 0;
            TCS_Data.Status = FSM_BUSY;
            Control_RGB_LEDs(1, 0, 0); // Red
            Delay(delay_duration);
            if (SWITCH_PRESSED(SW1_POS)) {
               Control_RGB_LEDs(1, 0, 1); // Magenta
               Delay(delay_duration);
               response_delay = 0;
               next_state = S2;
            } else {
               Control_RGB_LEDs(1, 1, 0); // Yellow
               Delay(delay_duration);
               next_state = S3;
            }
         } else {
            // stay in this state, awaiting a start command
         }
         break;
      case S2:
         Control_RGB_LEDs(0, 0, 1); // Blue
         Delay(delay_duration/2);
         Control_RGB_LEDs(1, 0, 0); // Red
         Delay(delay_duration/2);
         response_delay++;
         // Could update Count here for more frequent reporting
         if (SWITCH_PRESSED(SW2_POS))
            next_state = S3;
         else
            next_state = S2;
         break;
      case S3:
         for (flash_num = 0; flash_num < 3;
         flash_num++) {
            Control_RGB_LEDs(0, 1, 0); // Green
            Delay(delay_duration);
            Control_RGB_LEDs(0, 0, 0); // Off
            Delay(delay_duration);
         }
         next_state = S1;
         // Prepare return value
         TCS_Data.Count = response_delay;
         TCS_Data.Status = FSM_IDLE;
         break;
      default: next_state = S1;
         TCS_Data.Status = FSM_IDLE;
         TCS_Data.Count = 0;
         break;
   }
}
```

Another Task

```
void Task_Another_FSM(void) {
  static enum {S1,S2} next_state = S1;
  uint16_t n;

  switch (next_state) {
    case S1: // Wait for TCS to be idle and then start it
      if (TCS_Data.Status == FSM_IDLE) {
        TCS_Data.Delay = 1500;
        TCS_Data.Start = 1;
        next_state = S2;
      }
      break;
    case S2: // Wait for TCS to finish its work
      if ((TCS_Data.Status == FSM_IDLE) & (TCS_Data.Start == 0)) {
        // can now use TCS_Data.Count
        n = TCS_Data.Count;
        // use it for something
        next_state = S1;
      }
      break;
    default:
      next_state = S1;
      break;
    }
  }
```

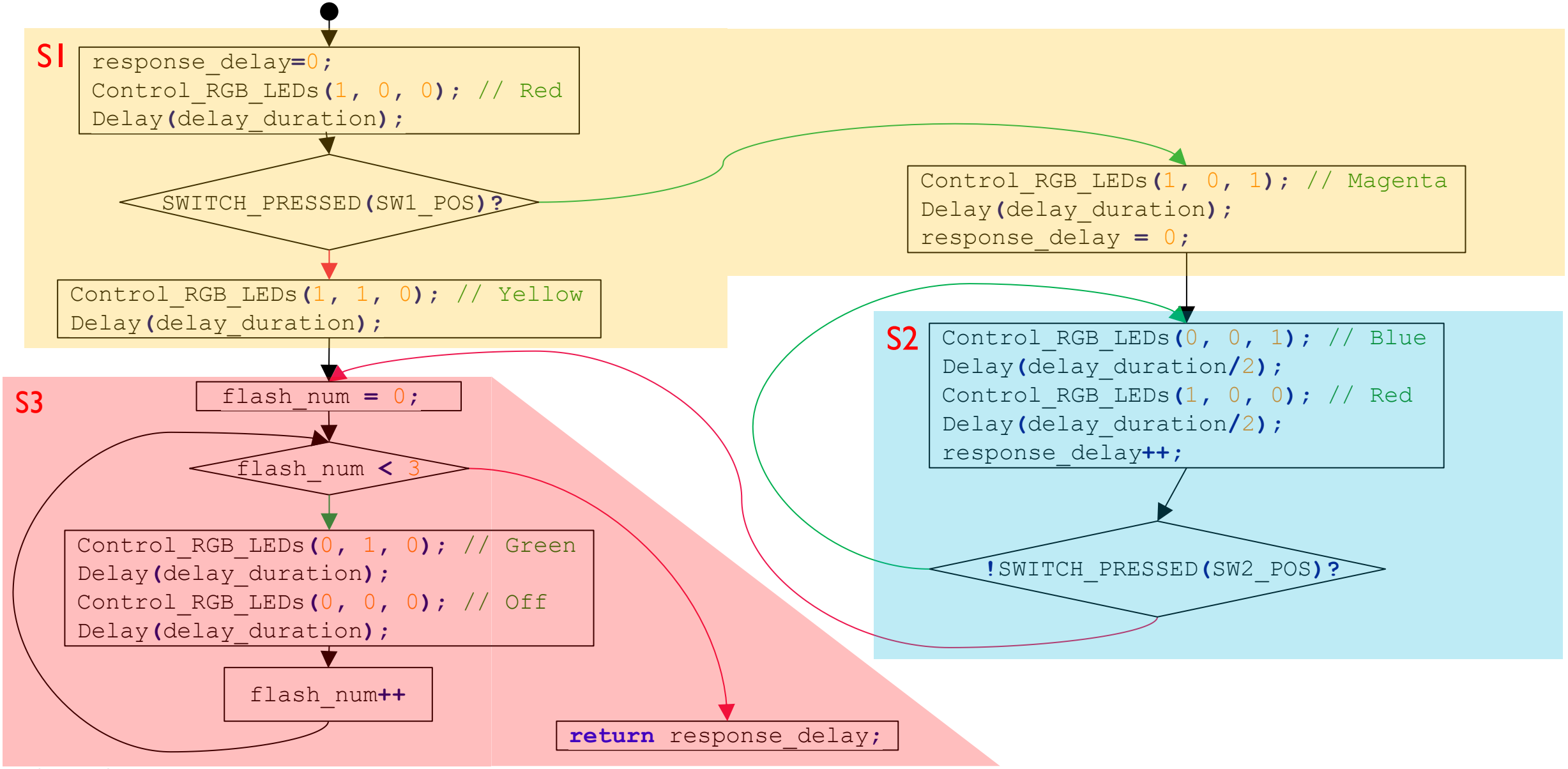
FSM Limitations

Splitting States and Granularity

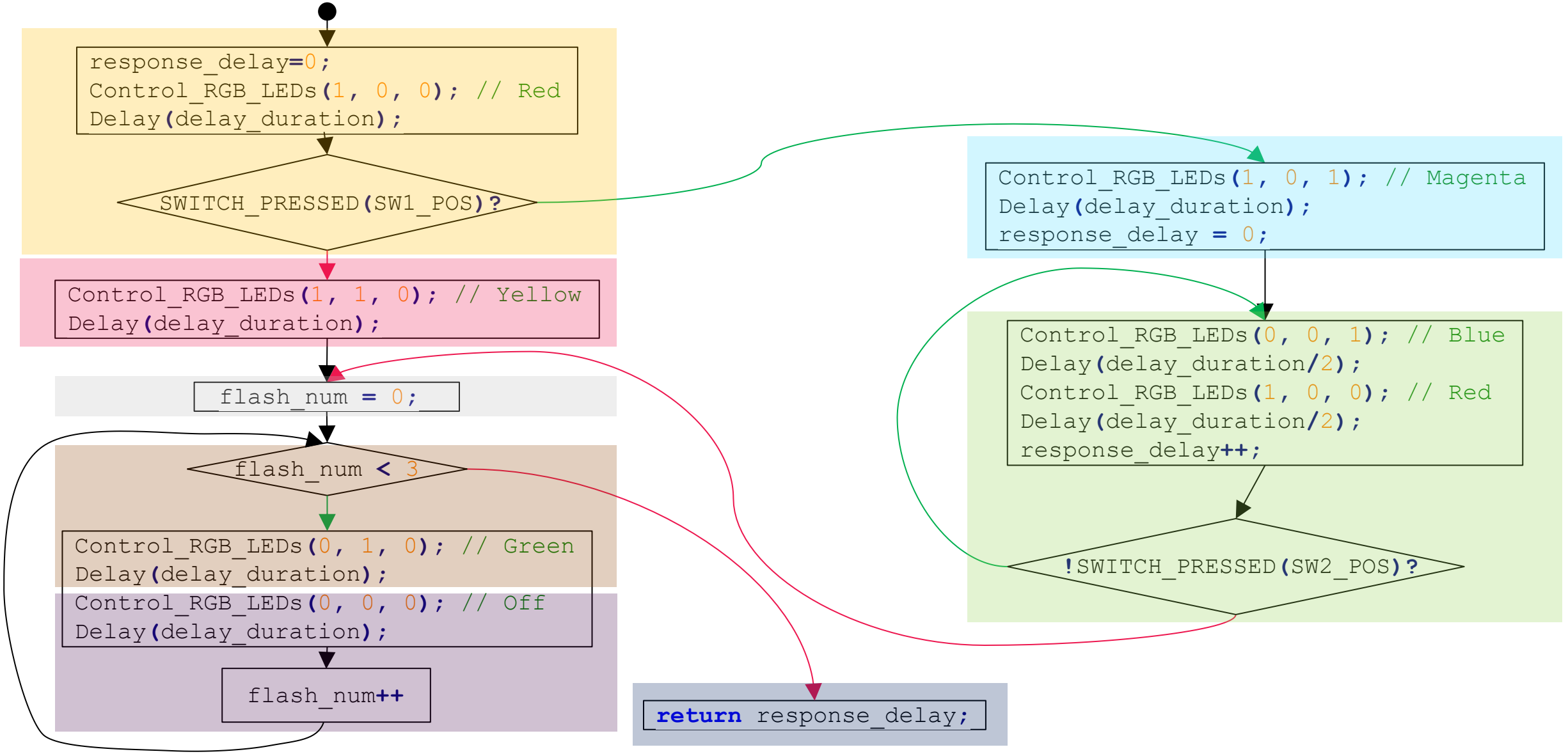


- FSM lets task function return earlier to scheduler
  - Introduces more frequent *scheduling points* (when scheduler can run)
- Limitations to state splitting
  - Subroutine calls
    - Easy to split code at light gray sections (between subroutine calls)
    - Other sections require splitting a subroutine call (e.g. Delay)
  - Control flow
    - Conditionals
    - Loops

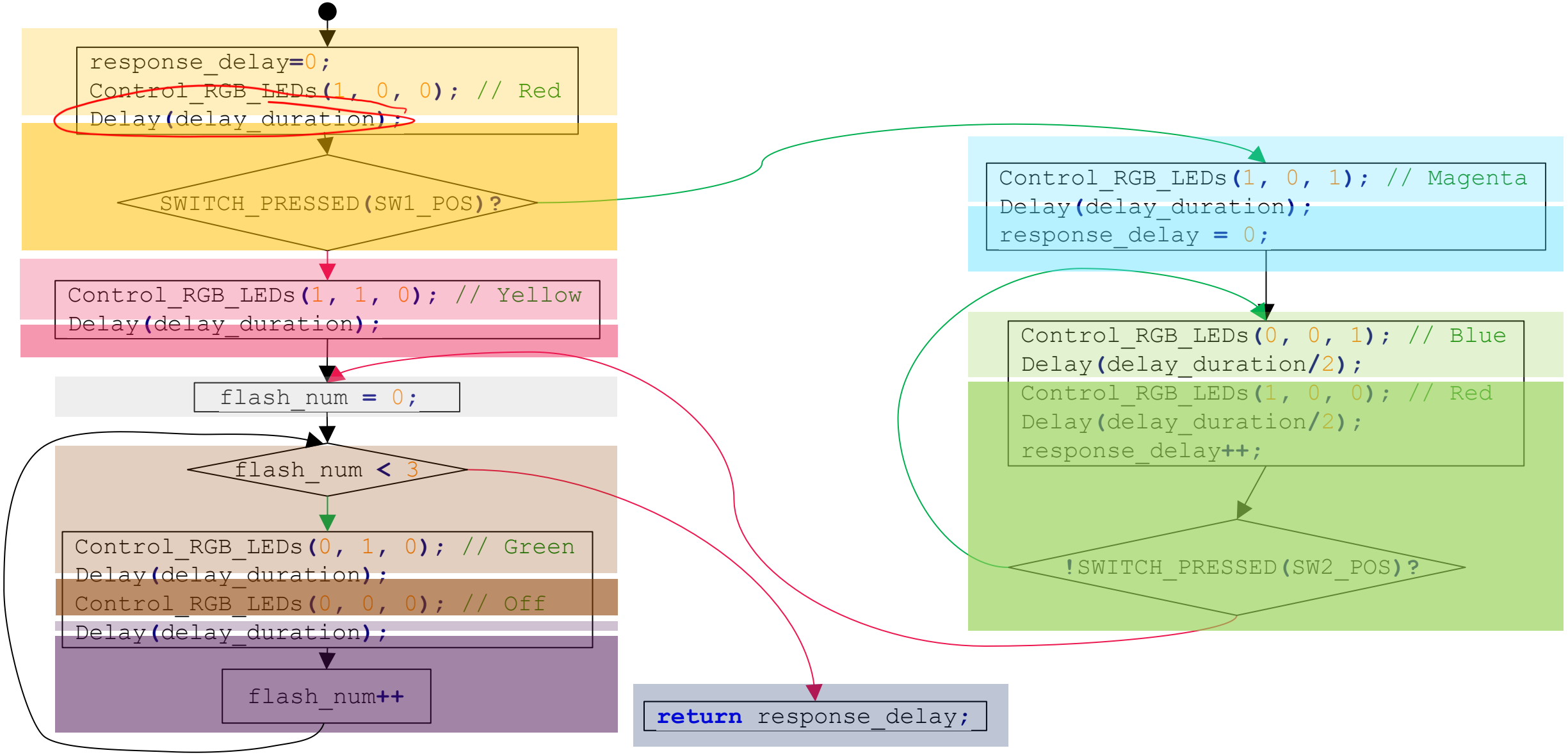
FSM with $T_{state} \leq 250$ ms Has 3 States



FSM with $T_{state} \leq 50$ ms Has 8 States

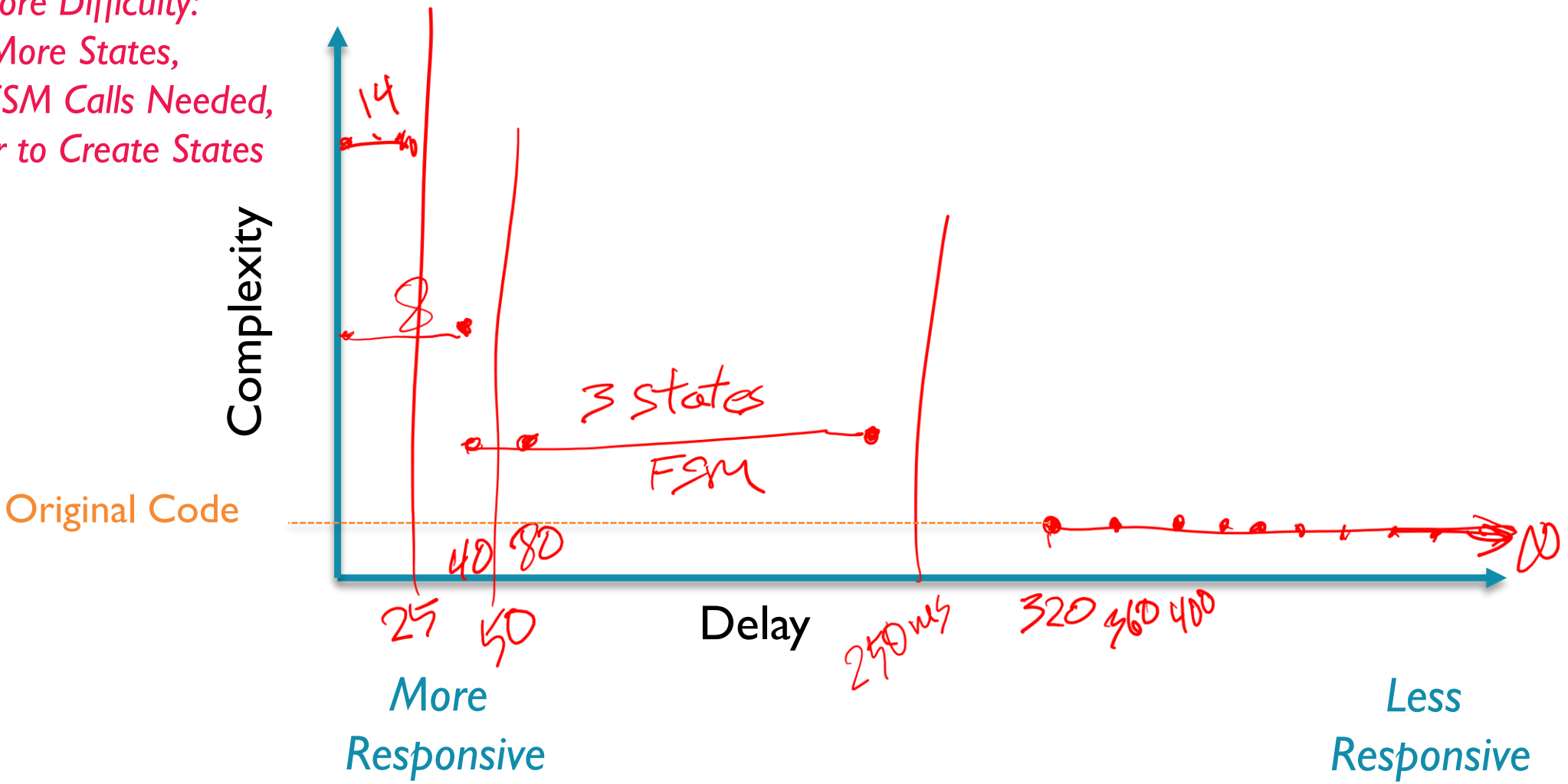


How to Make FSM with $T_{state} \leq 25$ ms?

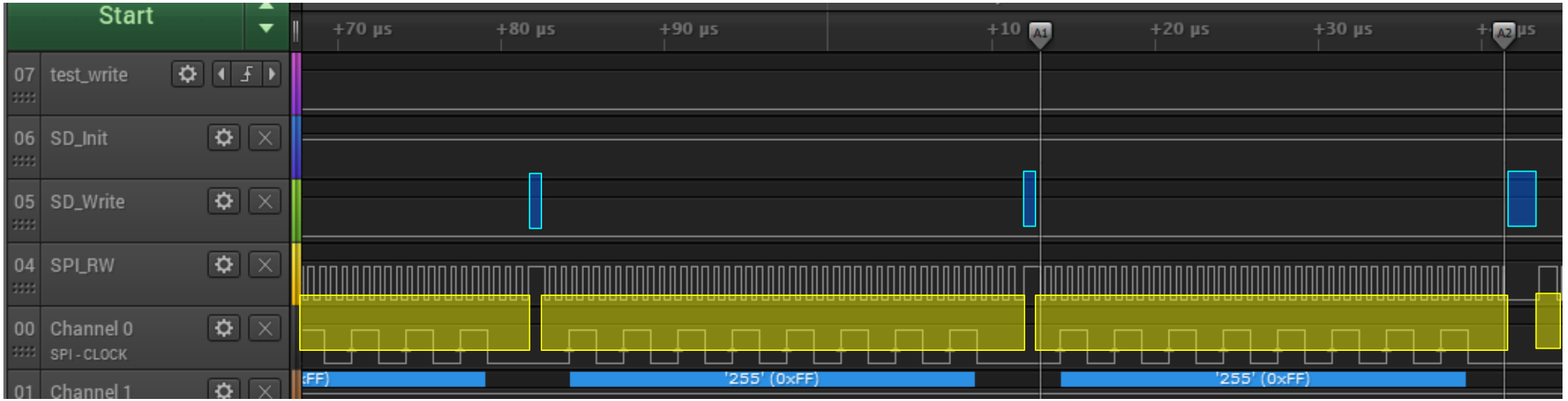


Scaling Limits of FSMs

More Difficulty:
More States,
More FSM Calls Needed,
Harder to Create States



Scaling Limits of FSMs



- What if task function calls a long, slow subroutine?
  - Example: SD_Init calls SD_Write with 300 kHz bus rate, each call takes 28 µs
  - Limits minimum state time, hurts responsiveness
- Options:
  - Inline subroutine into task function
  - Duplicate subroutine and split it
  - Re-architect code for early-out
  - Apply coroutines
  - Build hierarchical state machines
  - Let scheduler preempt running tasks at other scheduling points

More Complications of Inter-Task Communication

- In general case, tasks can run asynchronously: at any time relative to each other
- Questions to consider
  - Direction: Is the data flow one-way (message) or two-way (shared data)?
  - Buffering: Do we need to save each write, or is just the latest write needed?
  - Repeated Read: Is it ok for the reader to read the same value twice?
  - Notification: Do we need to notify the potential reader(s) when there is new data?
- Atomicity and data correctness
  - If writing to shared data object D takes multiple operations, then prevent all other accesses to D until last operation completes
- Will dive into these issues in preemptive scheduling module (next)

Advanced Topics

- Timing Analysis
  - Speed and progress: how many states need to execute in order to complete the work?
  - What is the maximum allowable delay between calls?

- Many FSM transformations possible
  - Merge states (e.g. w/replication) to improve speed
  - Integrate loop test to make a bottom-test
  - Replicate loop and split iterations to limit maximum state time (e.g. bail-out counter)

RTCS Task Releases

- Periodic or Event-Driven?
  - Periodic provides polling, limits maximum release frequency and shares processor with lower-priority tasks
  - Event-driven reduces response time, may not let lower-priority tasks run