```
/**
 * Marlin 3D Printer Firmware
 * Copyright (C) 2016 MarlinFirmware [https://github.com/MarlinFirmware/Marlin]
 *
 * Based on Sprinter and grbl.
 * Copyright (C) 2011 Camiel Gubbels / Erik van der Zalm
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 */
```

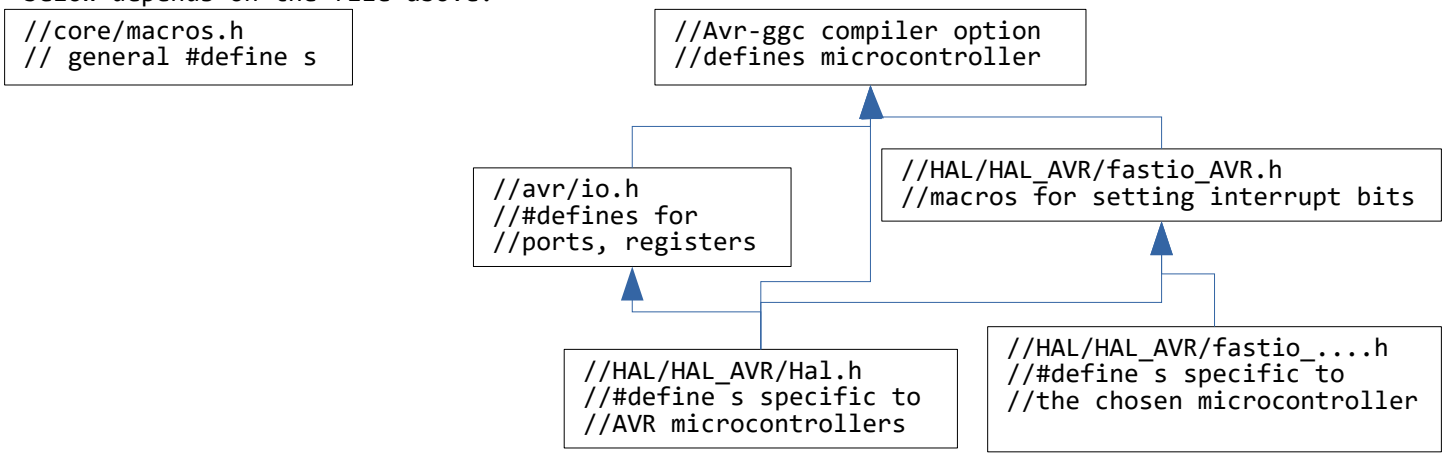
Marlin 2.0 code structure overview

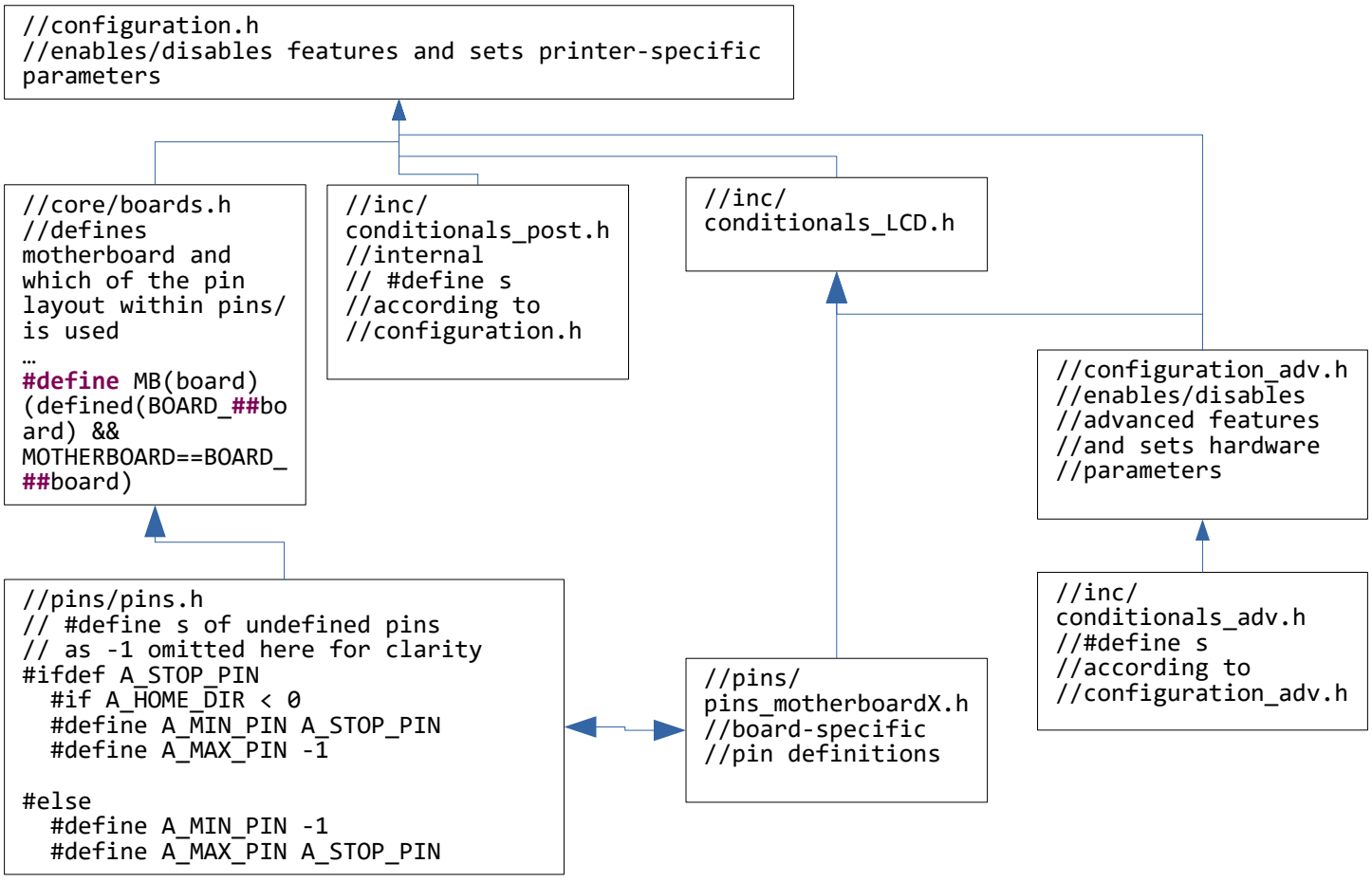
The following diagram gives a simplified overview of the code structure of Marlin firmware. It is showing the most important information flow for a minimalistic cartesian printer with minimal features enabled. It focuses on the two most vital commands: G28 (homing) and G1 (move).

For each object, the The module/file where it is defined is indicated. For a function func() that is declared in file foo.h, which is located in a subfolder moduleX/ the notation is e.g: moduleX/foo:func()

For objects (e.g. func2()) that are declared in a file that is located in the folder named “module/” (e.g. module/bar.h) the folder name is omitted:
bar:func2()

//Preprocessor directives
// This page gives an overview over files with basic #defines. An arrow indicates that the file below depends on the file above.





```
//This is running once upon printer (re)start:
Marlin:void setup() { // from Arduino.h, calls main(). Here the program starts.
  //printer startup routine
  pins:setup_killpin(); // from pins.h
  setup_printerhold();
  settings.load(); // load data from EEPROM if they exist
  planner:ZERO(current_position); // set current position to 0
  motion:sync_plan_position(); // from panner. Init planner/stepper equivalent for
                               // motion:current_position
  temperature:ThermalManager.init();
  print_job_timer_init();
  stepper:stepper_init();
  servo:servo_init();
  servo_probe_init(); // from probe
  endstops:endstops.enable_z_probe(false);
  stepper_indirection:enable_stepper_Drivers();
  SET_INPUT_PULLUP(HOME_PIN);
  lcd:lcd_init();
  lcd:lcd_reset_status();
}
```

```
// After setup(), this runs in an indefinite loop and is interrupted by inputs (thermometers,
// endstops, or G-code commands being sent to the buffer (via USB serial or from SD-card) or
  Because pulses have to be generated (e.g. inside stepper:stepper_ISR))
Marlin:void loop() {
  for (;;) { //repeat indefinitely
    if gecode/queue:commands_in_queue < BUFSIZE) {
      gcode/queue:get_available_commands() {
        if (gcode/queue:drain_injected_commands_P()) {return;}
      gcode/queue:get_serial_commands(); // reads serial and stores data in
                                         // serial_line_buffer
      };
    }
    gcode/queue:advance_command_queue(){
      if (!commands_in_queue) {return;}
      gcode/gcode:gcode.process_next_command() {
          parser:parser.parse(current_command);
          gcode/gcode:process_parsed_command(); // calls methods from the GcodeSuite namespace
                                                // for motion control, see page 5-8
    }
    endstops:endstops.event_handler();
    Idle(); // wait for input until timeout is reached. Calls core/parser.parser to react on
            // inputs. The parser calls the appropriate class or method:
  }
}
```

Interrupts (e.g. endstop triggered, see **page 4**)

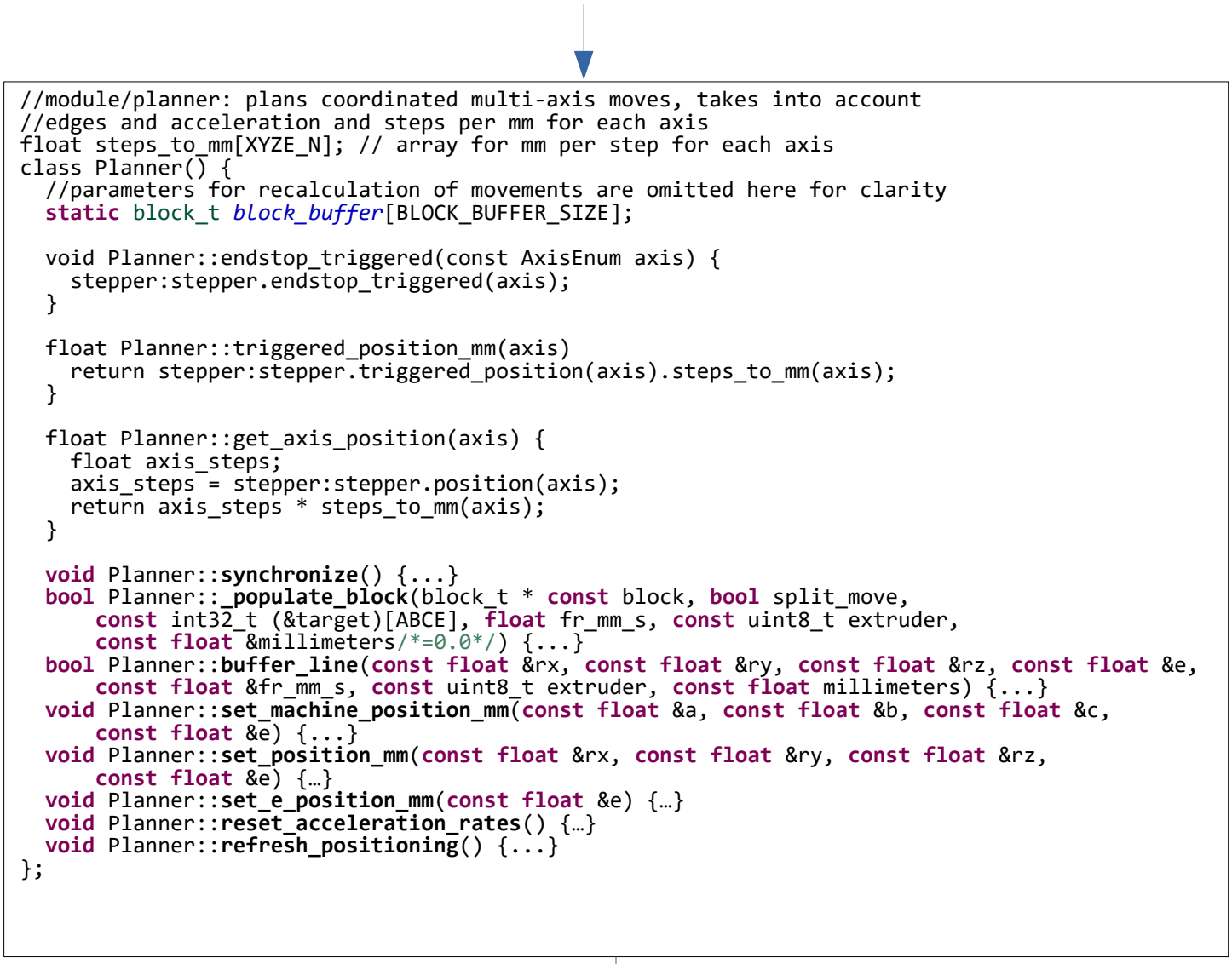
Inputs (see **page 4**)

```
// triggered endstops interrupt the current move if enabled
// HAL/HAL_AVR/endstop_interrups.h: endstop_ISR calls endstops:endstops.update() if an endstop
// was triggered

// module/endstops:
enum EndstopEnum // enum containing all possible endstops
class endstops() {
  typedef uint8_t esbits_t;

  private:
  static bool enabled, enabled_globally;
  static esbits_t live_state;
  static volatile uint8_t hit_state; // Use X_MIN, Y_MIN, Z_MIN and Z_MIN_PROBE as BIT index

  Public:
  void Endstops::poll() {
    #if DISABLED(ENDSTOP_INTERRUPTS_FEATURE)
      update();
    #endif
  }

  void Endstops::update() {
    // initial pin state checks are omitted here for clarity. Finally:
    // Record endstop was hit
    #define _ENDSTOP_HIT(AXIS, MINMAX) SBI(hit_state, _ENDSTOP(AXIS, MINMAX))

    // Call the endstop triggered routine for single endstops
    #define PROCESS_ENDSTOP(AXIS,MINMAX) do { \
      if (TEST_ENDSTOP(_ENDSTOP(AXIS, MINMAX))) { \
        _ENDSTOP_HIT(AXIS, MINMAX); \
        planner.endstop_triggered(_AXIS(AXIS)); \
      } \
    }while(0)
  }
};

extern Endstops endstops;
```

```
//inputs
//The following inputs can be sent to the buffer via USB serial or from SD-card:
//G28
gcode/calibrate/G28.cpp:void GcodeSuite::G28(const bool  always_home_all) {
  planner:planner.synchronize();
  motion:setup_for_endstop_or_probe_move()
  endstops:endstops.enable(true);
  motion:set_destination_from_current();
  const float z_homing_hight = (
    (gcode/parser:parser.seenval('R' ? gcode/parser:parser.value_linear_units() :
         Z_HOMING_HIGHT)
  );
  if (z_homing_hight && home_all || homeX || ...) {
    motion:destination[Z_AXIS] = z_homing_hight; // Z_AXIS is of type core/enum:AxisEnum
    if (motion:destination[Z_AXIS] > motion:current_position[Z_AXIS]) {
      motion:do_blocking_move_to_z(motion:destination[Z_AXIS]);
      }
    }
  if (home_all || homeX) {
    motion:homeaxis[X_AXIS]; // X_AXIS is of type core/enum:AxisEnum
  }
  if (home_all || homeY) {
    motion:homeaxis[Y_AXIS]; // X_AXIS is of type core/enum:AxisEnum
  }
  if (home_all || homeZ) {...}
  motion:sync_plan_position(); //sync with planner equivalent of current_position
  endstops:endstops.not_homing();
  motion:clean_up_after_endstop_or_probe_move();
  lcd:lcd_refresh();
  lcd:report_current_position()
}

//G0 or G1
gcode/motion/G0_G1.cpp: extern float destination[XYZE]
gcode/motion/G0_G1.cpp:void GcodeSuite::G0_G1() {
  if (is_running()) {
  motion:get_destination_from_command();
  motion:prepare_move_to_destination();
}
```

//Motion control from top (abstract) to bottom (bare metal registers for pin manipulation)

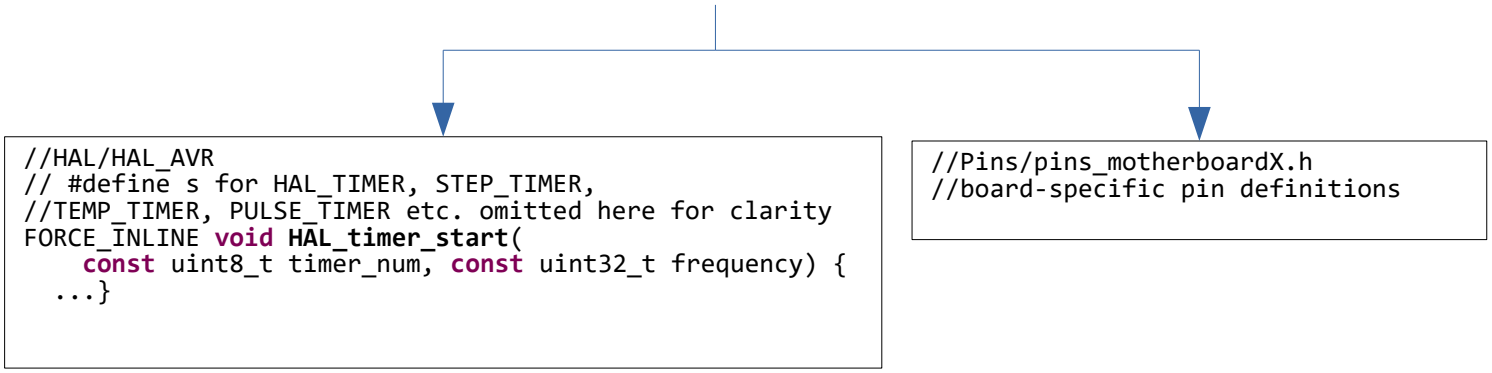
```
//module/motion:
// some variables omitted here for clarity
constexpr float soft_endstop_min[XYZE] = { X_MIN_BED, Y_MIN_BED, Z_MIN_POS, E_MIN_POS },
                soft_endstop_max[XYZE] = { X_MAX_BED, Y_MAX_BED, Z_MAX_POS, E_MAX_POS };

void report_current_position();

inline void set_current_from_destination() { COPY(current_position, destination); }
inline void set_destination_from_current() { COPY(destination, current_position); }

void get_cartesian_from_steppers();
void set_current_from_steppers_for_axis(const AxisEnum axis);

/**
 * sync_plan_position
 *
 * Set the planner/stepper positions directly from current_position with
 * no kinematic translation. Used for homing axes and cartesian/core syncing.
 */
void sync_plan_position();
void sync_plan_position_e();

/**
 * Move the planner to the current position from wherever it last moved
 * (or from wherever it has been told it is located).
 */
void line_to_current_position();

/**
 * Move the planner to the position stored in the destination array, which is
 * used by G0/G1/G2/G3/G5 and many other functions to set a destination.
 */
void buffer_line_to_destination(const float fr_mm_s);

void prepare_move_to_destination();

/**
 * Blocking movement and shorthand functions
 */
void do_blocking_move_to(const float rx, const float ry, const float rz, const float &fr_mm_s=0);
void do_blocking_move_to_x(const float &rx, const float &fr_mm_s=0);
void do_blocking_move_to_z(const float &rz, const float &fr_mm_s=0);
void do_blocking_move_to_xy(const float &rx, const float &ry, const float &fr_mm_s=0);

void setup_for_endstop_or_probe_move();
void clean_up_after_endstop_or_probe_move();

void bracket_probe_move(const bool before);
void setup_for_endstop_or_probe_move();
void clean_up_after_endstop_or_probe_move();

// homing
void set_axis_is_at_home(const AxisEnum axis);
void set_axis_is_not_at_home(const AxisEnum axis);
void homeaxis(const AxisEnum axis) {
  #define CAN_HOME(A) \
    (axis == _AXIS(A) && ((A##_MIN_PIN > -1 && A##_HOME_DIR < 0) || (A##_MAX_PIN > -1 && A##_HOME_DIR
  if (!CAN_HOME(X) && !CAN_HOME(Y) && !CAN_HOME(Z) && !CAN_HOME(E)) return;
  const int axis_home_dir = (home_dir(axis));
  planner:do_homing_move(axis, 1.5f * max_length(axis * axis_home_dir));

  // When homing Z with probe respect probe clearance
  const float bump = axis_home_dir * (home_bump_mm(axis));
  if (bump) {
    planner:do_homing_move(axis, -bump);
    // Slow move towards endstop until triggered
    planner:do_homing_move(axis, 2 * bump, get_homing_bump_feedrate(axis));
  }

  // For cartesian/core machines,
  // set the axis to its home position
  set_axis_is_at_home(axis);
  sync_plan_position();

  destination[axis] = current_position[axis];
}
```



```
//module/planner: plans coordinated multi-axis moves, takes into account
//edges and acceleration and steps per mm for each axis
float steps_to_mm[XYZE_N]; // array for mm per step for each axis
class Planner() {
  //parameters for recalculation of movements are omitted here for clarity
  static block_t block_buffer[BLOCK_BUFFER_SIZE];

  void Planner::endstop_triggered(const AxisEnum axis) {
    stepper:stepper.endstop_triggered(axis);
  }

  float Planner::triggered_position_mm(axis)
    return stepper:stepper.triggered_position(axis).steps_to_mm(axis);
  }

  float Planner::get_axis_position(axis) {
    float axis_steps;
    axis_steps = stepper:stepper.position(axis);
    return axis_steps * steps_to_mm(axis);
  }

  void Planner::synchronize() {...}
  bool Planner::_populate_block(block_t * const block, bool split_move,
      const int32_t (&target)[ABCE], float fr_mm_s, const uint8_t extruder,
      const float &millimeters/*=0.0*/) {...}
  bool Planner::buffer_line(const float &rx, const float &ry, const float &rz, const float &e,
      const float &fr_mm_s, const uint8_t extruder, const float millimeters) {...}
  void Planner::set_machine_position_mm(const float &a, const float &b, const float &c,
      const float &e) {...}
  void Planner::set_position_mm(const float &rx, const float &ry, const float &rz,
      const float &e) {…}
  void Planner::set_e_position_mm(const float &e) {…}
  void Planner::reset_acceleration_rates() {…}
  void Planner::refresh_positioning() {...}
};
```

```
//module/stepper_indirection
//if HAS_DRIVER == true, this module is an abstraction layer above stepper
```

```
//module/stepper
//translation from steps to pulses (controled via stepper_ISR
//flag all moving axes for proper endstop handling, check endstop limits
class Stepper {
  // several members omitted here for clarity
  public:
    // Constructor / initializer
    Stepper() { };
    // Initialize stepper hardware
    static void init();

    // Interrupt Service Routines
    // The ISR scheduler
    static void isr();
    // The stepper pulse phase ISR
    static void stepper_pulse_phase_isr();
    // The stepper block processing phase ISR
    static uint32_t stepper_block_phase_isr();

    // Check if the given block is busy or not - Must not be called from ISR contexts
    static bool is_block_busy(const block_t* const block);

    // Get the position of a stepper, in steps
    static int32_t position(const AxisEnum axis);
    // Report the positions of the steppers, in steps
    static void report_positions();

    // The stepper subsystem goes to sleep when it runs out of things to execute. Call this
    // to notify the subsystem that it is time to go to work.
    static void wake_up();
    // Quickly stop all steppers
    FORCE_INLINE static void quick_stop() { abort_current_block = true; }

    // The direction of a single motor
    FORCE_INLINE static bool motor_direction(const AxisEnum axis) {
        return TEST(last_direction_bits, axis); }
    // The last movement direction was not null on the specified axis.
    FORCE_INLINE static bool axis_is_moving(const AxisEnum axis) {...}
    // The extruder associated to the last movement
    FORCE_INLINE static uint8_t movement_extruder();

    // Handle a triggered endstop
    static void endstop_triggered(const AxisEnum axis);
    // Triggered position of an axis in steps
    static int32_t triggered_position(const AxisEnum axis);

    // Set the current position in steps
    static inline void set_position(const int32_t &a, const int32_t &b, const int32_t &c,
        const int32_t &e) {
      planner.synchronize();
      const bool was_enabled = STEPPER_ISR_ENABLED();
      if (was_enabled) DISABLE_STEPPER_DRIVER_INTERRUPT();
      _set_position(a, b, c, e);
      if (was_enabled) ENABLE_STEPPER_DRIVER_INTERRUPT();
    }
    static inline void set_position(const AxisEnum a, const int32_t &v) {
      planner.synchronize();
      #ifdef __AVR__
        // Protect the access to the position. Only required for AVR, as
        //  any 32bit CPU offers atomic access to 32bit variables
        const bool was_enabled = STEPPER_ISR_ENABLED();
        if (was_enabled) DISABLE_STEPPER_DRIVER_INTERRUPT();
      #endif
      count_position[a] = v;
      #ifdef __AVR__
        // Reenable Stepper ISR
        if (was_enabled) ENABLE_STEPPER_DRIVER_INTERRUPT();
      #endif
    }

    // Set direction bits for all steppers
    static void set_directions();
};
```


//HAL/HAL_AVR
// #define s for HAL_TIMER, STEP_TIMER,
//TEMP_TIMER, PULSE_TIMER etc. omitted here for clarity
FORCE_INLINE void HAL_timer_start(
const uint8_t timer_num, const uint32_t frequency) {
...}
//Pins/pins_motherboardX.h
//board-specific pin definitions