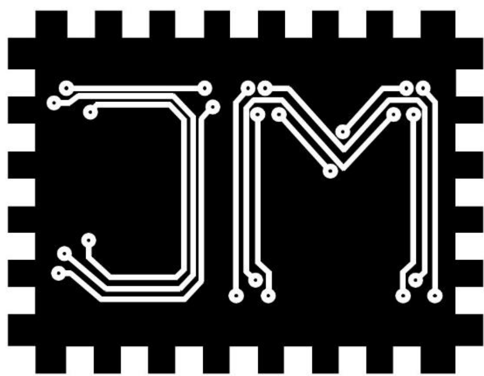


NMSW02 – Flight Manager Events

Comments	Revision	Date	Author
Initial Release	A	January 9, 2025	J. Mays

1. References

1. NMSW01 – Flight Manager Sequencer
2. NMSW03 – Flight Manager Estimator

2. Purpose

This document describes the EVENTS software capability as illustrated in Simulink, and then further discussed in C/C++ software. Original implementation was written in Simulink and then derived into C/C++ software for compilation onto hardware. Custom Simulink and C/C++ libraries were created to align the two languages.

3. Design Description

The EVENTS SWC is responsible for estimating events that are expected to occur. EVENTS uses states from the SEQUENCER [1] and ESTIMATOR [2] to derive flight events: liftoff detection, solid rocket motor (SRM) burnout, reentry, touchdown, pad abort and flight abort. This software works in parallel with the SEQUENCER [1] to flag when things occur.

4. Interface Control Document

The EVENTS SWC input and output buses are shown below. This SWC is called by the FLIGHT_MANAGER, and its elements are populated to the vehicle state vector every cycle count.

Table 1: EVENTS input bus

App	Direction	Hierarchy	Element	DataType	Rows	Columns	Comment
fm	out	seq	segment	uint16_t	1	1	FM sequencer state
fm	out	est.state	accel_sensed_up	float	1	1	Sensed accel in the up direction with the UEN frame
fm	out	est.state	alt_dot	float	1	1	Vertical velocity
fm	out	est.state	tilt	float	1	1	Vehicle tilt with the vertical axis

Table 2: EVENTS output bus

App	Direction	Hierarchy	Element	DataType	Rows	Columns	Comment
fm	out	events	liftoff	bool	1	1	Liftoff event indicator
fm	out	events	burnout	bool	1	1	SRM burnout event indicator
fm	out	events	reentry	bool	1	1	Reentry event indicator
fm	out	events	touchdown	bool	1	1	Touchdown event indicator
fm	out	events	pad_abort	bool	1	1	Pad Abort limits have been exceeded
fm	out	events	flt_abort	bool	1	1	Flight Abort limits have been exceeded

5. Pre-Configured Gains

Each SWC has an initialization subroutine that initializes parameterized gains. The gains are first defined in Matlab, and then are autocoded into a Cpp file for reference during compilation of the Cpp version of this SWC. The following code snip shows the gain subroutine for EVENTS.

```
function [gains] = init_gains_events(config)
% Events SWC gains

% Load common gains
common = init_gains_common();

%% 2nd order filter for accel state
gains.filter.wn   = 10*C_HZ;
gains.filter.zeta = sqrt(2)/2;
```

```
% Compute coeffs for direct form 2 version (needed for Cpp code)
s = tf('s');
w = gains.filter.wn;
z = gains.filter.zeta;
G = (w)^2 / (s^2 + 2*w*z*s + w^2);
Gz = c2d(G, common.timing.dt, 'tustin');
gains.filter.coeffs.a0 = Gz.Numerator{1}(1);
gains.filter.coeffs.a1 = Gz.Numerator{1}(2);
gains.filter.coeffs.a2 = Gz.Numerator{1}(3);
gains.filter.coeffs.b1 = -Gz.Denominator{1}(2); % neg
gains.filter.coeffs.b2 = -Gz.Denominator{1}(3); % neg

%% Liftoff
gains.liftoff.acc_up_threshold   = 11 * C_M/C_SEC/C_SEC;
gains.liftoff.persistent_cycles  = uint32((0.02 * C_SEC) / common.timing.dt);

%% Pad Abort
gains.pad_abort.tilt_limit = 2.0 * C_DEG; % FLT-300
gains.pad_abort.acc_up_ll   = (9.8 - 0.5) * C_M/C_SEC/C_SEC;
gains.pad_abort.acc_up_ul   = (9.8 + 0.5) * C_M/C_SEC/C_SEC;
gains.pad_abort.timeout_cycles = uint32((0.01 * C_SEC) / common.timing.dt);

%% Flight Abort
gains.flt_abort.tilt_limit     = 30.0 * C_DEG; % In flight tilt limit
gains.flt_abort.timeout_cycles = uint32((0.01 * C_SEC) / common.timing.dt);

%% Touchdown
gains.touchdown.alt_rate_max   = -3.0 * C_M/C_SEC;
gains.touchdown.timeout_cycles = uint32((0.2 * C_SEC) / common.timing.dt);

%% Reentry
% NA

%% SRM Burnout
% Load our models used to parameterize the sim. And use nominal MC index
mc = SimCore.MonteCarlo(0);
[~, srm_tunable] = loadSRMParameters(mc, config);

% Use the SRM thrust and time vector to derive when the SRM should
% theoretically be out of thrust with some safety factor
start_ii = find(srm_tunable.thrust_lookup > 0, 1, 'first');
end_ii = find(srm_tunable.thrust_lookup > 0, 1, 'last');
start_time = srm_tunable.time_lookup(start_ii);
end_time = srm_tunable.time_lookup(end_ii);

% Add 50% scale factor and round up to the nearest second
sf = 1.5;
gains.burnout.srm_timeout_cycles = uint32(ceil((end_time - start_time) * sf) / common.timing.dt);

% Add accel lower threshold
gains.burnout.acc_up_threshold = 0.0 * C_M/C_SEC/C_SEC;
gains.burnout.up_accel_timeout_cycles = uint32((0.02 * C_SEC) / common.timing.dt);

end
```

6. Software Logic

A high-level capture of the `FLIGHT_MANAGER` software logic is shown in Figure 1. The `EVENTS` software capability is highlighted.

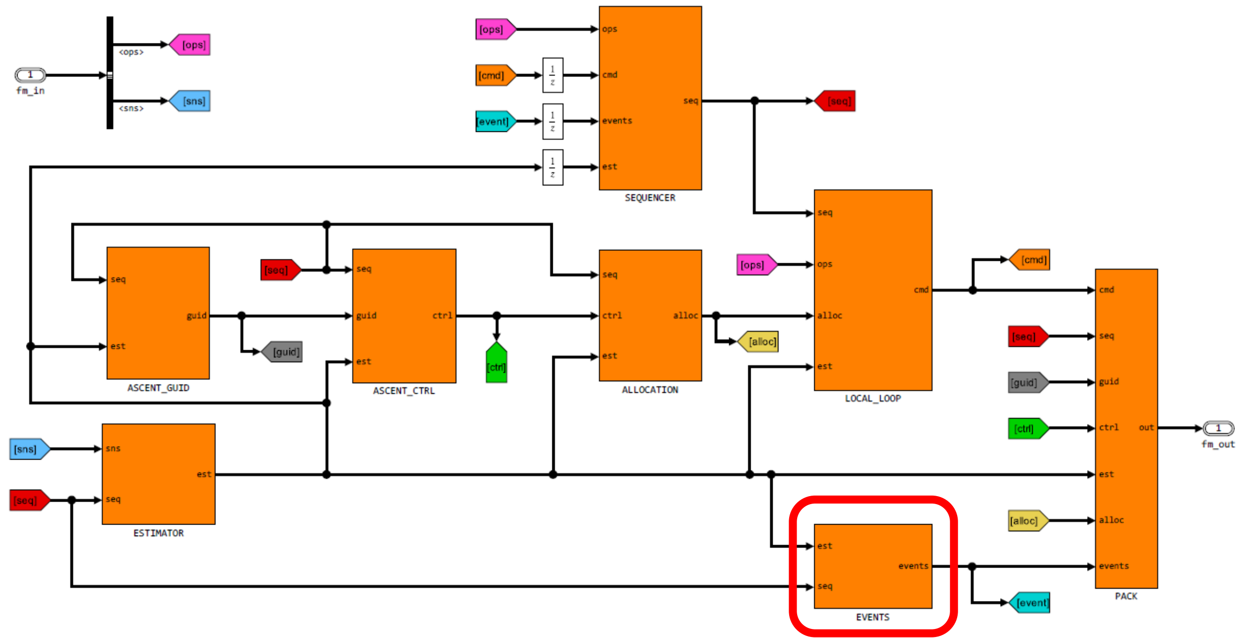


Figure 1: Flight Manager

The EVENTS software capability is shown in Figure 2. This software uses custom libraries, namely a 2nd order discrete filter, persistence, limits, and latch blocks to help derive the output elements.

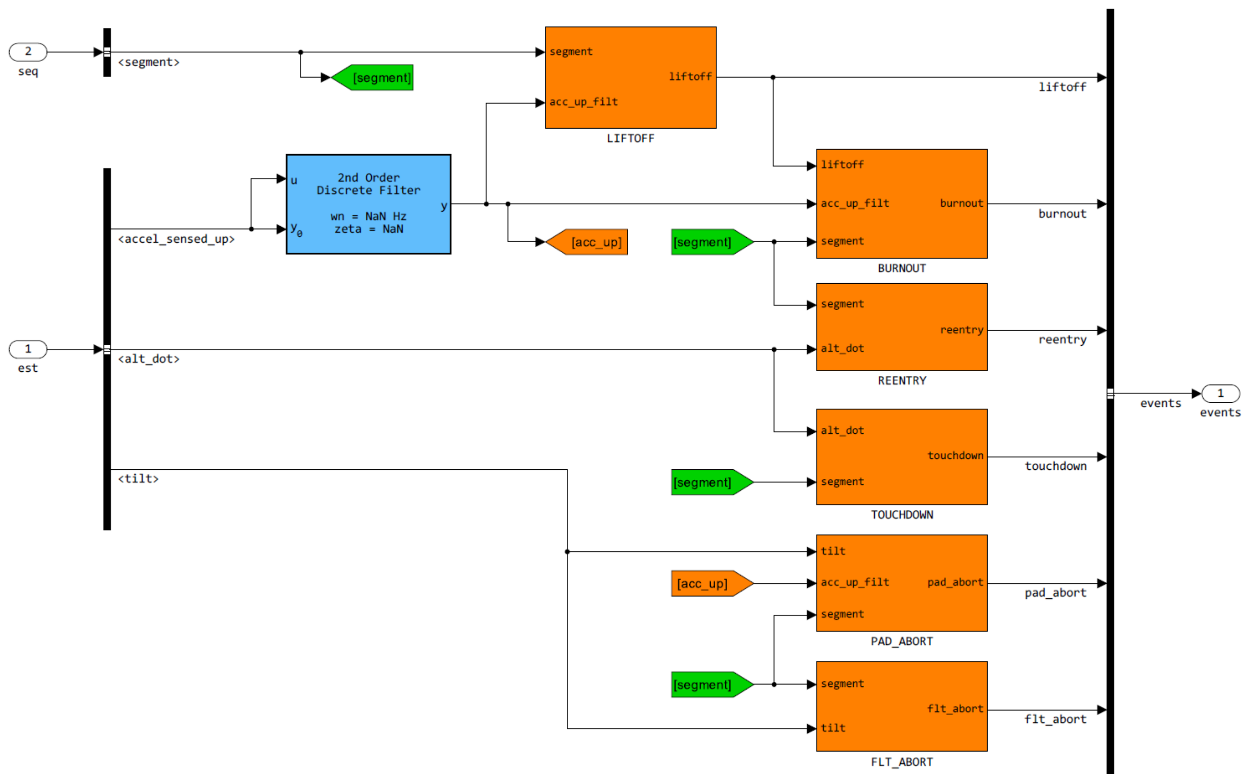
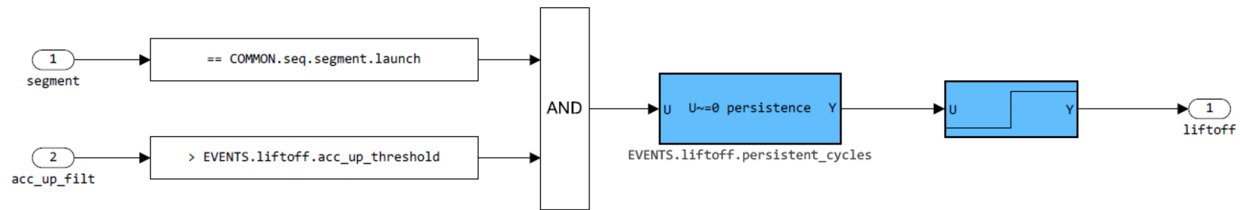


Figure 2: EVENTS software capability

Each event's subsystem Simulink and C/C++ software logic is shown below. Note that in-house versions of the same software were created such that the translation between Simulink and C would be trivial.

6.1. Liftoff

The liftoff SW checks our filtered upward acceleration while we are in the Launch segment to detect if liftoff has occurred.



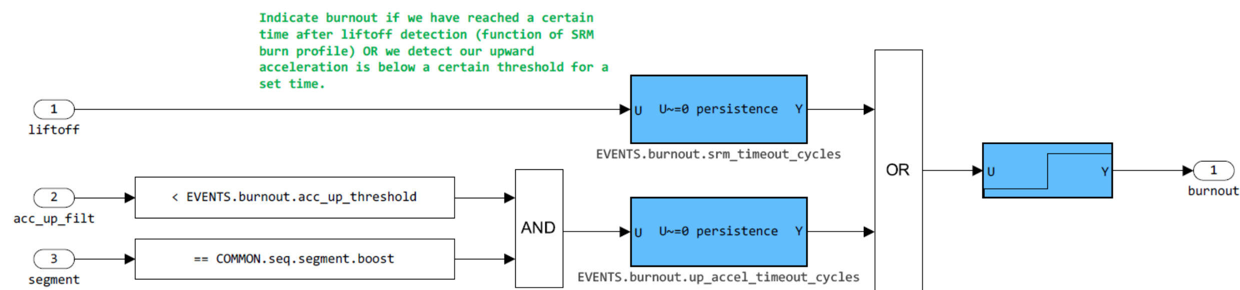

```
// Filter up acceleration through a 2nd order LPF
acc_up_filtered_ = acc_2ndorder_filter.update(bus_fm_out_est.state.accel_sensed_up);

// LIFTOFF
// Latch the liftoff event if the filtered accel is above a limit for a set persistence and we are in the LAUNCH
segment
in_launch_                    = bus_fm_out_seq.segment == COMMON.seq.segment.launch;
above_liftoff_acc_            = acc_up_filtered_ > GAINS.liftoff.acc_up_threshold;
persist_above_liftoff_thresh_ = liftoff_persistence.exec(above_liftoff_acc_ && in_launch_);
bus_fm_out_events.liftoff     = liftoff_latch.exec(persist_above_liftoff_thresh_);
```

Figure 3: Liftoff

6.2. Burnout

We detect solid rocket motor burnout by either a clear lack of vertical acceleration (indicating we are no longer thrusting), or a timeout from after we have lifted off. This timeout is determined using the known thrust profile out of the rocket motor along with some scaling factor so that we would never erroneously set burnout until thrust is actually zero.

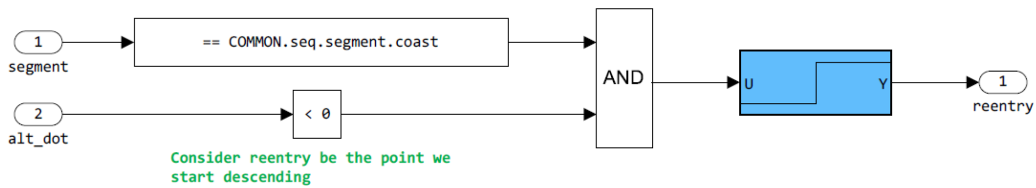



```
// BURNOUT
// Latch the burnout event after launch if our sensed accel is below a threshold, or a timer (fcn of SRM) runs out
in_boost_                = bus_fm_out_seq.segment == COMMON.seq.segment.boost;
srm_burnout_backup_      = burnout_srm_persistence.exec(bus_fm_out_events.liftoff);
below_acc_up_threshold_  = acc_up_filtered_ < GAINS.burnout.acc_up_threshold;
acc_burnout_primary_     = burnout_acc_persistence.exec(below_acc_up_threshold_ && in_boost_);
bus_fm_out_events.burnout    = burnout_latch.exec(acc_burnout_primary_ || srm_burnout_backup_);
```

Figure 4: Burnout

6.3. Reentry

Once we are in Coast and our estimate of vertical velocity is negative, we latch reentry.

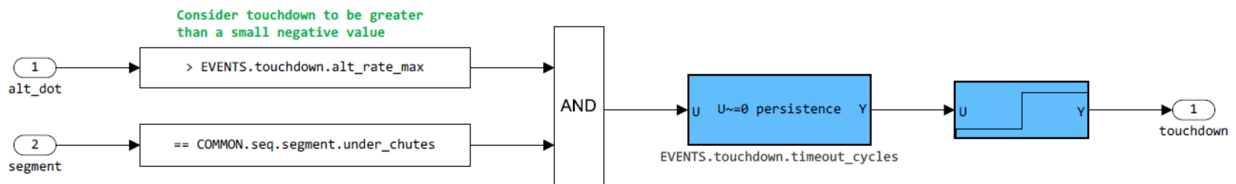



```
// REENTRY
// Latch reentry if we are in COAST and the estimated vertical velocity is negative
in_coast_       = bus_fm_out_seq.segment == COMMON.seq.segment.coast;
is_descending_  = bus_fm_out_est.state.alt_dot < 0.0;
bus_fm_out_events.reentry = reentry_latch.exec(in_coast_ && is_descending_);
```

Figure 5: Reentry

6.4. Touchdown

If chutes are out and our descent rate is above a small negative value, then we latch touchdown.

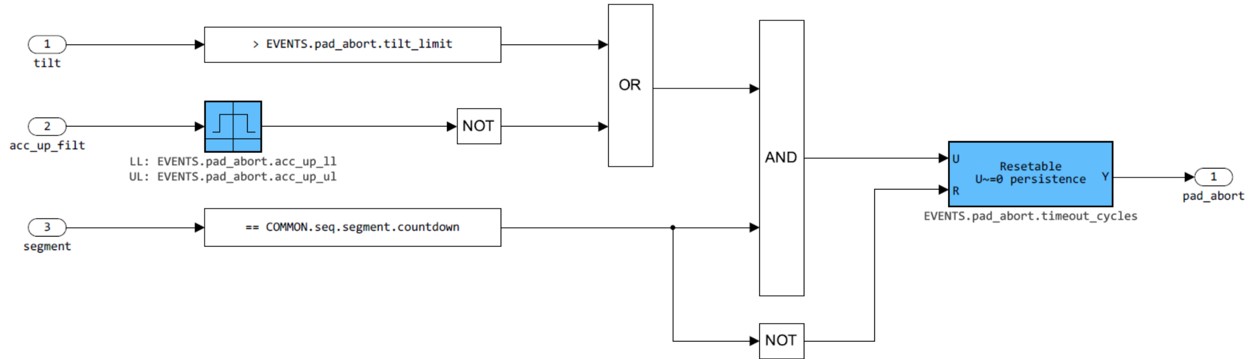



```
// TOUCHDOWN
// Latch touchdown if we are under chutes and our descent velocity is low
in_under_chutes_    = bus_fm_out_seq.segment == COMMON.seq.segment.under_chutes;
is_not_moving_      = bus_fm_out_est.state.alt_dot > GAINS.touchdown.alt_rate_max;
persist_touchdown_  = touchdown_persistence.exec(in_under_chutes_ && is_not_moving_);
bus_fm_out_events.touchdown = touchdown_latch.exec(persist_touchdown_);
```

Figure 6: Touchdown

6.5. Pad Abort

During the Countdown segment, we can pad abort. The Pad Abort software is meant to check for things that would gate the pyro signal to the solid rocket motor. If the estimated tilt from vertical beyond a certain amount, or if our vertical accel is not within the expected range, then we will abort while on the pad.

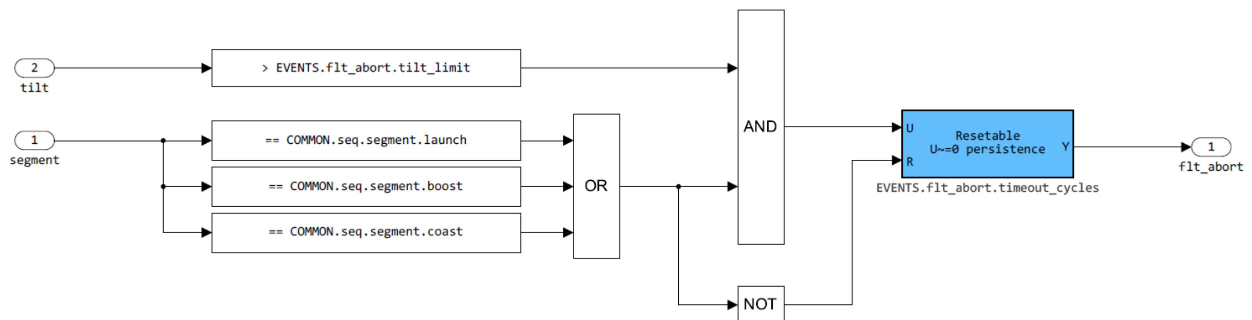



```
// PAD ABORT
// Send abort flag if our filtered accel is outside the expected interval OR our tilt is beyond a certain limit during
// the countdown segment. Reset the persistence fcn if we are not in the countdown segment
in_countdown_        = bus_fm_out_seq.segment == COMMON.seq.segment.countdown;
outside_acc_limits_  = !pad_acc_up_interval.exec(acc_up_filtered_);
outside_tilt_limit_  = bus_fm_out_est.state.tilt > GAINS.pad_abort.tilt_limit;
limits_exceeded_     = outside_acc_limits_ || outside_tilt_limit_;
bus_fm_out_events.pad_abort = pad_abort_reset_persistence.exec(limits_exceeded_ && in_countdown_, !in_countdown_);
```

Figure 7: Pad abort

6.6. Flight Abort

A flight abort is possible while in a flight segment. Should the estimated tilt angle from vertical be above a certain value (of while is much larger than the guidance trajectory), then the `EVENTS` SWC will issue a flight abort, and the vehicle will deploy the chutes to stop the vehicle from flying away from the launch pad. This is a pseudo flight termination system.




```
// FLT ABORT
// Send abort flag if our tilt is beyond a certain limit during flight segments. Reset the persistence fcn if we are
not
// in the flight segments anymore.
outside_tilt_limit_         = bus_fm_out_est.state.tilt > GAINS.flt_abort.tilt_limit;
in_flight_                  = bus_fm_out_seq.segment == COMMON.seq.segment.launch ||
                              bus_fm_out_seq.segment == COMMON.seq.segment.boost ||
                              bus_fm_out_seq.segment == COMMON.seq.segment.coast;
bus_fm_out_events.flt_abort = flt_abort_reset_persistence.exec(outside_tilt_limit_ && in_flight_, !in_flight_);
```

Figure 8: Flight abort

Software Validation & Unit Tests

> ***Justification for Lack of Unit Test Development:*** *This project is* ***for-fun****. It is not meant to represent the actions that would be taken in a fully staffed and/or an especially human safety critical aerospace project/program. Rather than a long and cumbersome unit test development and documentation process of each SWC, desktop simulations were used to provide evidence of sufficient behavior out of this particular SWC. While not all possible code coverage avenues are explicitly checked in the nominal monte-carlo simulations, all requirements are enforced in post processing and elevated to the user should they not be met. This should be acceptable for the level of rigor required for this project.*

The following plots are shown from either a nominal simulation or a one-off test created by the author. These plots should give the reader confidence that the SWC behaves as intended.