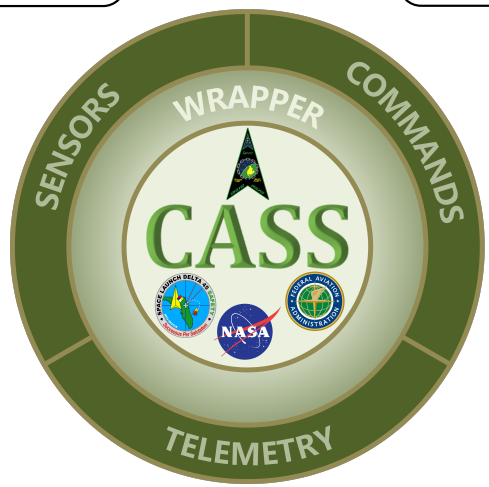
# Core Autonomous Safety Software (CASS) Operational Release (OR) Comparison

Jeffrey Cherry (SLD 30 SEAE)
jeffrey.cherry.1@spaceforce.mil

Cass Russett (SLD 30 SEAE)
richard.russett@spaceforce.mil



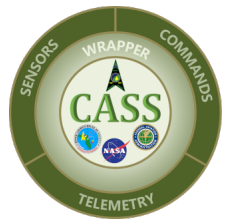
# Bottom Line ...

## Which CASS Operational Release should I use?

CASS OR3

## Why?

1. Errors/deficiencies in OR1 and OR2 are corrected.
2. Selectable vacuum IP or drag-corrected IP.
3. Drag-corrected IP configurable per-region.
4. Utilize host vehicle data via vehicle sensor.
5. Define new tracking sensor variables.
6. User-defined computations.
7. Built-in functions.
8. Three-way tables.
9. Efficient moving boundaries.
10. Real-time Green Time rule.
11. Support tool enhancements.



# Pros and Cons ...

## CASS OR1 Update 01

Traceable, rule-based evaluation of safety criteria.

Fast vacuum impact point computation.

Contains known errors; longitude range violation and table reuse error.

Need to duplicate Mission Rules to apply same rule to each sensor.

Voting creates additional rules, with large complex expressions.

Limited to ARM/DESTRUCT command and DISABLE FTS command.

Each Mission Rule is tied to one command; rule results cannot be shared between commands.

DISABLE FTS command state not accessible via API function. Must be retrieved from database.

Large API; 35 functions. Mix of mission and database functions; poor cohesion.

Exorbitant waste of memory for strings.

Use of moving gates is not recommended.

"Primary" sensor notational convenience is often mistaken with "best" sensor.

Frame data evaluated for only one sensor.

## CASS OR2

Traceable, rule-based evaluation of safety criteria.

Fast vacuum impact point computation.

Rule templates. Major simplification in writing rules.

Complete voting system. Major simplification in writing rules.

Separation of rules and decisions. Rules no longer tied to commands.

Extensible command system; multiple commands triggered independently.

New rules section for staging events.

Frame data computed for each sensor.

Data interface separated from API; API reduced to 20 functions.

Data interface capabilities expanded.

Dynamic boundaries and static gates.

New sensor data items.

Contains known errors; table reuse error and violation of two requirements.

Dynamic boundary movement computation is inefficient.

Dynamic boundary definition in Mission Rules is overly-complicated and disjointed.

## CASS OR3

Traceable, rule-based evaluation of safety criteria.

Fast vacuum impact point computation.

Drag-corrected impact point computed in user-defined regions with user-definable per-region tuning parameters, wind models, and atmosphere models.

User-defined drag models and drag model switching conditions.

Selectable impact computation; vacuum impact only provides OR2 computational speed.

User-defined computations per rule: copy database values; call user-defined table functions; call built-in utility functions.

User-defined tracking sensor variables.

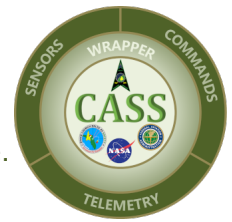
New vehicle data interface with user-defined ingestible data items.

Improved dynamic boundary movement computations.

Uncomplicated dynamic boundary definitions.

Real-time boundary dependent Green Time Rule.

. . . and more!



# Section/Subsection Hyperlinks, Part 1



# Section/Subsection Hyperlinks, Part 2

2 CASS OR2

- Longitude Bound Error Over Anti-Meridian
- Noncompliance with requirements CASS-SRS-0949 and CASS-SRS-1075
- Inconsistency in Gate Rule Processing
- Flight Software Highlights
- Static Gates and Dynamic Boundaries
- Template Rules and Complete Voting System
- Extensible Commands
- Separation of Rules and Decisions
- Memory, Reference Frames, New Data Items, and Data Interface

# Section/Subsection Hyperlinks, Part 3

3 CASS OR3

- Unnecessary Include File
- Incorrect Stream Length
- Inconsistency in Gate Rule Processing
- Flight Software Highlights
- Drag-Corrected Impact Point Estimate
- Enhanced Computational Capabilities
- Sensor-Related Changes
- Persist Attribute Removed
- Dynamic Boundaries
- Real-Time Boundary-Dependent Green Time Rule
- Miscellaneous

# CASS Operational Release 1 (OR1)

## Known Error

A known deficiency within the vacuum impact point algorithm is that invalid impact points can be computed when the vehicle location is below the geoid surface (i.e., at negative altitude) and the apogee never exceeds the geoid surface.

## Outcome

**Work-Around:** Add a condition in Mission Rules to verify height is above the geoid surface for rules dependent on an impact point. If height is at or below the geoid surface, use the present position rather than impact position for the rule.
**Correction:** The vacuum impact point algorithm was enhanced to locate the impact point at the geoid surface above the vehicle location when both vehicle position and apogee position are below the geoid surface. SpaceX requested the enhanced code and was provided CASS OR1 Update 01 in March 2016. After passing delta IV&V, unit testing, and formal testing, CASS OR1 Update 01 Flight Software was recommended for use in safety critical applications.

# CASS Operational Release 1 (OR1)

## Known Error

When a computed vacuum impact point estimate begins falling on one side of the anti-meridian and impacts on the other side of the anti-meridian, the computed impact longitude could exceed the -180° to +180° range. The longitude value is correct, just outside the "normalized" range.

## Outcome

Impact is Mission Rule dependent and only for missions expecting to cross the anti-meridian. SLD 30 can provide corrected code on request, but recertification of the revised CASS Flight Software is the responsibility of the user.

# CASS Operational Release 1 (OR1)

## Known Error

When a mission rule references the same table multiple times, the first reference to the table will produce correct results while subsequent references to the same table may produce invalid results.

## Outcome

Create duplicate tables rather than use the same table multiple times. SLD 30 can provide corrected code on request, but recertification of the revised CASS Flight Software is the responsibility of the user.

# CASS Operational Release 1 (OR1)

## CASS OR1 Flight Software Highlights

- Original release was certified in July 2015.
- Update 01 certification effort, funded by Range User, completed in 2016.
- Original release and Update 01 are 13,460 Source Lines Of Code (SLOC).
- Conforms to C++ 2003 and C++ 2011 language standards.
- CASS Flight Software provides an operational database and execution environment for a "Mission Rules" script, driven by ingesting real-time data from range tracking sensors.
- CASS Flight Software computes vacuum instantaneous impact predictions for each tracking sensor, among many other data items.
- Single flight termination command controlled by `Terminate` class rules.
- Single FTS disable database flag controlled by `Safing` class rules.
- Flight termination command timing defined in `Settings` section.
- Reference frame computations apply to a "primary" sensor only.

## Wrapper Responsibilities

- Control MDL storage location and access protocol.
- Create an `MDL_Tree` object with wrapper-controled access to MDL content.
- Use `MDL_Tree` object to create CASS Flight Software `Master` object. `Master` object will:
  - Read MDL content via wrapper-controlled access function.
  - Validate MDL load using 16-bit CRC.
  - Validate MDL based on format and content.
  - Establish an execution environment for MDL contents.
  - Provide status to wrapper of `Master` object creation process.
- Register tracking sensors with `Master` object by providing access to each tracking sensor's data area.
- Retrieve health and status information from `Master` object for load verification and assurance checks.

## Wrapper Responsibilities (Continued)

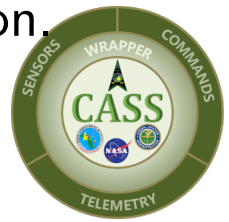
- Initiate hazardous operations via `Master` object built-in test function.
- Initiate rule processing via `Master` object logic enable function.
- Provide `Master` object with dual hardware discrete logic flags indicating when liftoff occurs.
- Cyclically call `Master` object `Update` function.
- Retrieve messages after `Update` function completes.
- Retrieve command states (ARM, DESTRUCT) from `Master` object functions.
- Retrieve the `haveRulesSafed` flag from `Master` object database.
- Output telemetry data items.
- Initiate appropriate action, if any.

## Mission Rules File

- Extensible Markup Language (XML) file conforming to Range Safety Operations Markup Language (rsoML) Schema Definition.
- Composed of 10 major sections:
  - `Mission`: Defines mission name, date, launch point.
  - `UserDefines`: Create named floating point variables.
  - `Settings`: Define mission parameters.
  - `NavSensors`: Define tracking sensors, qualify logic, and priority list.
  - `Stages`: Define thrust/burnout event pairs.
  - `ReferenceFrames`: Define origins for reference frames.
  - `Boundaries`: Define static boundaries and their vertices.
  - `Tables`: Define one-way or two-way linear interpolation functions.
  - `MissionRules`: Define both termination and safing criteria.
  - `StreamSets`: Define telemetry output data items.

## MissionRules Section

- List of rules.
- Three different types of rules:
  - `GenericRule`: Contains an `ApplyWhen` conditional expression and an optional class-specific conditional expression.
  - `GateRule`: Contains an `ApplyWhen` conditional expression, an optional class-specific conditional expression, a reference point (latitude and longitude of vehicle subpoint or estimated impact point), and static gate definition (gate endpoint locations, trip mode, and cross persit).
  - `MapBoundaryRule`: Contains an `ApplyWhen` conditional expression, an optional class-specific conditional expression, a reference point (latitude and longitude of vehicle subpoint or estimated impact point), and the name of a reference boundary defined in the `Boundaries` section.



## MissionRules Section (Continued)

- Two different classes of rules:
  - `Terminate`: Defines flight termination via a `FireWhen` conditional expression.
  - `Safing`: Defines FTS safing via a `SafeTerminateRulesWhen` conditional expression.
- Any rule may contain an optional `Tables` section.
  - A list of user-defined table function calls.
  - Functionally equivalent to $r = f(a)$ or $r = f(a, b)$ for one-way tables and two-way tables, respectively.
  - Table name (i.e., $f()$) must be defined in the `Tables` section.
  - Inputs (i.e., "$a$" or "$a, b$") must be floating point database variables or constants.
  - Output (i.e., $r$) must be a floating point database variable.
  - Gate endpoint coordinates may be redefined via a table call. A feature called a **moving gate**.

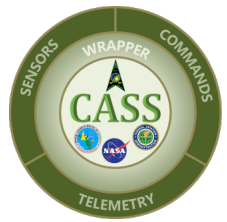
## Conditional Expressions

- Boolean conditions are enclosed within `<cond>` *term operator term* `</cond>` elements.
- Terms are integer, floating point, time, or Boolean variables or constants defined in the database.
- Terms in a Boolean condition must be type identical; two integers, two floating point, two times, or two Boolean values.
- For integer, floating point, or time conditional expressions, the HTML less than (`&lt;`) or HTML greater than (`&gt;`) operators must be used.
- For Boolean conditional expressions, the "`is`" operator must be used (`<cond>` *flag* `is false </cond>`).
- Boolean conditional expressions may use the negation element (`<not>` *flag* `is true </not>`).

**Conditional Expressions (Continued)**

- Compound Boolean conditions can be created using either <and/> or <or/> operators.
- Boolean conditions may be nested within <cond> ...</cond> elements.
- The simple majority <vote> operator can be used to vote over multiple Boolean conditions, such as:

```
<vote>
  <cond> GpsA.accelTotal &lt; 4.0 </cond>
  <cond> GpsB.accelTotal &lt; 4.0 </cond>
  <cond> ImuA.accelImuX  &lt; 4.0 </cond>
</vote>
```



## Conditional Expressions (Continued)

- Conditional expressions are used in the following:
  - `QualifyLogic` expressions used when defining a tracking sensor within the `NavSensors` section.
  - `IgnitionLogic` expressions used when detecting a thrust event for a `Stage` definition within the `Stages` section.
  - `BurnoutLogic` expressions used when detecting a burnout event for a `Stage` definition within the `Stages` section.
  - `ApplyWhen` expressions used in any rule definition within the `MissionRules` section.
  - `FireWhen` expressions used in any `Terminate` class rule definition within the `MissionRules` section.
  - `SafeTerminateRulesWhen` expressions used in any `Safing` class rule definition within the `MissionRules` section.

# CASS Operational Release 2 (OR2)

## Known Error

When a mission rule references the same table multiple times, the first reference to the table will produce correct results while subsequent references to the same table may produce invalid results.

## Outcome

Create duplicate tables rather than use the same table multiple times. SLD 30 can provide corrected code on request, but recertification of the revised CASS Flight Software is the responsibility of the user.

# CASS Operational Release 2 (OR2)

## Known Error

CASS OR2 Flight Software incorrectly computes tracking sensor variable **NoValidDataTime** such that it does not fully comply with requirement SRS-CASS-0949 and incorrectly computes tracking sensor variable **isValidSensorData** such that it does not fully comply with requirement SRS-CASS-1075. Affects Mission Rule expressions referencing either variable and any subsequent operations that depend on those expression's result.

## Outcome

**Work-Around:** Use alternative logic in Mission Rules to replicate the per requirement content for both **isValidSensorData** and **NoValidDataTime** values. Alternatively, use **isGoodSensorData** flag instead of **isValidSensorData** flag and use **NoGoodDataTime** value instead of **NoValidDataTime** value. SLD 30 can provide corrected code on request, but recertification of the revised CASS Flight Software is the responsibility of the user.

# CASS Operational Release 2 (OR2)

## Known Error

Gate Rules are templates expanded over a set of tracking sensors. The Gate Rule detects when a reference point (i.e., the tracking sensor's vehicle subpoint or its estimated impact point) moves through a gate. Due to a logic inconsistency in the CASS Flight Software gate processing code, a gate crossing may go undetected in rare circumstances.

## Outcome

The effect of this error can only be realized in extraordinarily rare circumstances when a reference point projects on the downrange side of the gate line and is within a small error tolerance (e.g., 50 millimeters) of the gate line. Should this unique error occur, severity depends on the Mission Rules. In a worst case scenario, not detecting a gate crossing may lead to an inadvertent termination of a good vehicle or the inability to terminate a vehicle flying dangerously.

# CASS Operational Release 2 (OR2)

## CASS OR2 Flight Software Highlights

- Original release was certified in August 2018.
- Original release is 16,201 Source Lines Of Code (SLOC).
- Conforms to C++ 2003 and C++ 2011 language standards.
- Static gates and dynamic boundaries.
- Template rules added to rsoML and Mission Rules XML files.
- Complete voting system (quorum, abstain, inquorate options, tie resolution).
- Voting operations are restricted to event detection and commands.
- Extensible commands using either immediate or stair-step algorithms.
- New rules section for detecting staging events (thrust/burnout pairs).
- Event detection, commands, and rules are logically separated.
- Memory run-time requirement drastically reduced.
- Reference frames apply to each tracking sensor.
- Data interface and wrapper API separated.
- New per-tracking-sensor data items.

# CASS Operational Release 2 (OR2)

## Static Gates

- Moving gates in OR1 were used to implement Chevrons.
- Moving gate algorithm found to contain vulnerabilities.
- Moving gate extension dropped; only static gates are allowed in OR2.

## Dynamic Boundaries

- User-defined table functions used to compute boundary vertex movement.
- Not all boundary vertices need to move. User can define which boundary vertices are static and which boundary vertices move.
- Supports Chevrons (expanding boundary) or landing (collapsing boundary).

## Template Rules and Complete Voting System

- Simplifies rule writing and reduces rule set complexity compared to OR1.
  - Reduced number of rules to write by $1/n$, where $n$ is the number of user-defined tracking sensors.
  - Need for generic "voting" rules eliminated by complete voting system, further reducing the number of rules to write.
- Rule "template" is replicated for each user-defined tracking sensor.
  - `Subset` element allows user to specify a restricted set of tracking sensors.
- Rule reference points are limited to either vehicle subpoint or predicted impact point for a tracking sensor.
- Tracking sensor field names are used instead of qualified field names.
  - `latIP` rather than `GPS_A.latIP`, for example.

## Template Rules and Complete Voting System (Continued)

- Syntax of rules optimized for voting.
  - Optional `ApplyWhen` expression replaced with required `InvalidWhen` expression. `InvalidWhen` expressions evaluating to `true` indicate the rule will abstain from a vote.
  - Optional `FireWhen` and `SafeTerminateRulesWhen` expressions replaced with required `Result` expression. `Result` expression outcome used as ballot for a vote.
- Vote operators used for detecting events based on multiple inputs.
  - Thrust event detection using `IgnitionLogic` and `BurnoutLogic` expressions.
  - Detecting command state changes using `CommandWhen` expressions.
  - Decision expressions may use `<and/>` or `<or/>` operators and `<cond> ...</cond>` elements in addition to the `<vote>` operator.
- Vote operator arguments are template or non-template rule names $\Rightarrow$ rules vote on decisions.

## Template Rules and Complete Voting System (Continued)

- Complete voting system.
  - Ballots are the Boolean value from a rule's `Result` expression outcome.
  - Vote operator outcome is a Boolean value.
  - Three non-abstaining rules define a quorum.
  - Rule abstains when rule's `InvalidWhen` expression outcome is `true`.
  - Attributes of vote operator define behavior when vote lacks a quorum (i.e., inquorate vote) or the vote is tied.

| Attribute Name | Description | Values |
|---|---|---|
| `zero` | Vote outcome when all rules abstain. | `true, false` |
| `one` | Vote outcome for one non-abstaining rule. | `true, false, ballot-result` |
| `two` | Vote outcome for two non-abstaining rules. | `true, false, or, and` |
| `tie` | Vote output for a tie vote. | `true, false` |

## Commands

- New `Commands` section of Mission Rules XML file. Used to define each `Command`.
- New commands must be supported by AFTU developer.
- Each command can move from one state to the next, where the first state is `COMMAND_OFF`.
- Movement from one state to the next is controlled by a `CommandWhen` expression.
- Two state-based command "algorithms" available:
  - **Immediate command:** Once `CommandWhen` expression evaluates to `true`, the command transitions from the `COMMAND_OFF` state to the `COMMAND_ACTIVATE` state and latches. Emulates a "Safing" command from OR1.
  - **Stair Step command:** When `CommandWhen` expression evaluates to `true`, the command transitions along the path
    `COMMAND_OFF` $\rightarrow$ `COMMAND_READY` $\rightarrow$ `COMMAND_ACTIVATE`.
    State transitions are reversible when `CommandWhen` expression evaluates to `false`;
    `COMMAND_ACTIVATE` $\rightarrow$ `COMMAND_READY` $\rightarrow$ `COMMAND_OFF`.

    Emulates a "Terminate" command from OR1.

## Commands (Continued)

- Each Stair Step command manages an `ActivateSignal` time duration value. It is incremented by the `Update` cycle rate (i.e., database parameter `SamplePeriod`) for each `Update` cycle the `CommandWhen` expression evaluates to `true`, and decremented by the `Update` cycle rate each time the `CommandWhen` expression evaluates to `false`.
- Each command using a Stair Step algorithm manages the following timing parameters. These parameters were global in OR1. In OR2, these parameters are bound to a specific command and independent of any other command.
  - `TimeToReady` – The time for `ActivateSignal` to transition `COMMAND_OFF` $\rightarrow$ `COMMAND_READY` or `COMMAND_READY` $\rightarrow$ `COMMAND_OFF`.
  - `TimeToActivate` – The time for `ActivateSignal` to transition `COMMAND_READY` $\rightarrow$ `COMMAND_ACTIVATE` or `COMMAND_ACTIVATE` $\rightarrow$ `COMMAND_READY`.
  - `TimeCap` – The the upper bound for `ActivateSignal`, which is restricted to the range [0.0, `TimeCap`].
  - `TimeSlop` – The time tolerance for `ActivateSignal`.

## New Stage Rules Section

- New `Rules` subsection in the `Stages` section of the Mission Rules XML file.
- Only template or non-template `GenericRules` allowed in the stage rules subsection.
- Stage rules aid in writing decision expressions for staging events, i.e., `IgnitionLogic` and `BurnoutLogic` expressions.

## Rules Provide Data for Decisions

- Rules are user-defined computations to generate data.
- Decisions ascertain a course of action based on data.
- OR1 conjoined rules (data generation) with flight termination and FTS disabling (i.e., safing) decisions. OR2 separates this unrelated associated between rules and decisions.
- OR2 pools raw tracing sensor data, derived tracking sensor data (e.g., impact point estimate), rule outcomes, and data based on previous decisions (e.g., time since ignition). Decision expressions can be written based on this data pool.
- Database information can be used directly in `<cond> ...</cond>` elements and rule outcomes can be used directly in `<vote>` operators, with voting policy defined explicitly via `<vote>` operator attributes.

## Reduced Memory Usage

- General use of strings was removed for OR2.
- Specific use of strings for reading MDL data, converting string-to-numeric representation, and storing string constants (e.g., Mission name string, file revision string) was retained.
- OR2 introduced proxies (unsigned 32-bit integers) to identify unique objects. OR1 used unique string names for the same purpose.
- Proxies use 4 bytes per identifier, OR1 strings use more than 50 bytes per string identifier.
- Depending on Mission Rules size and complexity, OR2 run-time memory requirements were reduced to half the memory requirement used by OR1 for an equivalent set of Mission Rules.

**Note**

The `MDL_Tool` generates a proxy-name map file, in CSV format, for use by GSE to convert proxy values to corresponding names from the Mission Rules. Proxy values were added to outgoing messages in OR2 to identify the associated object. OR1 lacks the ability to identify the source object's name in the outgoing message.

# CASS Operational Release 2 (OR2)

## Reference Frames

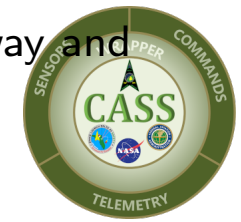
- User-defined reference frames provide a means to simplify rule expressions when measurement or threshold values are naturally expressed in a reference frame other than the standard EFG reference frame.
  - A position and velocity state can be transformed to a new reference frame when given the new frame's origin position and orientation relative to the standard EFG reference frame.
  - Reference frames are frequently used to transform tracking sensor information to downrange and crossrange coordinate systems.
- OR2 transforms each tracking sensor's state vector to projected position and velocity parameters for each user-defined reference frame. In contrast, OR1 performed this transformation for only one selected tracking sensor's state vector.
- Transformed parameters for each user-defined reference frame are part of each tracking sensor's derived data and available for use in both template and non-template rules.
- Tracking sensor position and velocity vectors are transformed to each reference frame.
- Impact point position and velocity vectors are transformed to each reference frame.

## New Per-Tracking-Sensor Data Items

- Residual impact point height.
- Impact point state data (position, velocity, and velocity magnitude).
- Orbital energy.
- Altitude rate.
- Data flags with "valid" and "good" distinction.
- Data refreshed flag.

## Data Interface

- In OR1, interface functions for read-only access to the CASS Flight Software database were mixed in with the `Master` class, though they were unrelated to the `Master` class' primary purpose of supporting the Safety function.
- The database interface functions were moved into a separate `Data_Interface` class for OR2, which reduced the number of functions defined in the `Master` class by half (complexity reduction).
- The `Data_Interface` class enhanced and expanded access to additional data items.
  - Read-only access to the CASS Flight Software database based on proxy or fast-access index.
  - Access provided lists of Sensors, Boundaries, Tables, Reference Frames, Stages, Commands, Stage Rules, and Mission Rules.
  - Read-only access to boundary attributes and vertices.
  - Access to boundary functions to determine if a reference point is interior or exterior to the boundary and the distance of a reference point to the nearest boundary edge.
  - Access to frame evaluation function that provides the frame data produced from transforming a given position vector and velocity vector.
  - Access to table attributes and table interpolation functions for both one-way and two-way tables.
  - Read-only access provided for the `Update` call counter.



# CASS Operational Release 3 (OR3)

## Known Error

There is an unnecessary `include` directive in file `Named_Object.cpp` that pulls in the standard library `iostream` header. The `iostream` library is not used by the `Named_Object` class nor any of its descendant classes. Since the code does not use the `iostream` library, the effect is compiler and linker dependent – some compilers and linkers may include the unused library code into the final executable image while others my not. Regardless, the software has been verified compliant with all requirements establishing that the unnecessary `include` directive has no effect on Flight Software behavior.

## Outcome

Since there is no effect on code behavior or performance, users may safely use the CASS OR3 Flight Software code base as is. The unnecessary directive will be removed from the code base if an update or revision of the CASS OR3 Flight Software becomes necessary.

# CASS Operational Release 3 (OR3)

## Known Error

When defining a stream in the Mission Rules, the `Rules_Checksum()` function is supposed to insert the 16-bit unsigned integer checksum of the original Mission Rules XML file, but instead a 32-bit unsigned integer value is inserted into the stream. The effect is that each occurrence of function `Rules_Checksum()` within a stream definition, increases the length of the stream by 2 bytes more than expected. If not accounted for, ground processing of the stream data will not extract the checksum value correctly and all subsequent measurements will also be processed incorrectly.

## Outcome

The fault lies in the CASS OR3 Flight Software, though it has no effect on the software's functionality. Users may safely use the CASS OR3 Flight Software code base as is. A correction will be scheduled for the next update of the CASS OR3 Flight Software. Till this occurs, users should account for the extra bytes in the ground processing of the stream data.

# CASS Operational Release 3 (OR3)

## Known Error

Gate Rules are templates expanded over a set of tracking sensors. The Gate Rule detects when a reference point (i.e., the tracking sensor's vehicle subpoint or its estimated impact point) moves through a gate. Due to a logic inconsistency in the CASS Flight Software gate processing code, a gate crossing may go undetected in rare circumstances.

## Outcome

The effect of this error can only be realized in extraordinarily rare circumstances when a reference point projects on the downrange side of the gate line and is within a small error tolerance (e.g., 50 millimeters) of the gate line. Should this unique error occur, severity depends on the Mission Rules. In a worst case scenario, not detecting a gate crossing may lead to an inadvertent termination of a good vehicle or the inability to terminate a vehicle flying dangerously.

## CASS OR3 Flight Software Highlights

- Original release was certified in July 2022.
- Original release is 28,786 Source Lines Of Code (SLOC).
- Conforms to C++ 2003 and C++ 2011 language standards.
- Selectable drag-corrected impact point estimate.
  - Enabled (default) or disabled (vacuum impact only) via Mission Rules "Settings".
  - Drag-corrected impact point computations performed in user-defined regions.
  - Drag-corrected algorithm parameters defined per user-defined region (tunable).
  - Uses U.S. Atmospheric Model (1976) standard as built-in atmospheric model or can be overridden per-region with a user-defined atmospheric model.
  - Uses per-region user-defined wind models with day-of-launch wind data updates.
  - User-defined drag models or user-defined ballistic coefficients.
  - Switching drag models based on used-defined expressions (flight event dependent).
- Enhanced computation capabilities.
  - "Tables" section in rules replaced with "Compute" section.
  - Allows copying one database entity's value to another database variable.
  - Uses function syntax to call user-defined table functions.
  - Added user-defined three-way tables.
  - Provides over 20 built-in-functions.

## CASS OR3 Flight Software Highlights (Continued)

- User-defined tracking sensor variables.
- User-defined vehicle sensor data items.
- Tracking sensor filters.
- Removed "`persist`" attribute from expressions.
- Improved dynamic boundary vertex movement computations.
- Improved dynamic boundary definition syntax in Mission Rules.
- Real-time boundary-dependent Green Time Rule.
- New reference frame parameter.
- Replaced 16-bit CRC with 32-bit CRC.

## Other CASS OR3 Highlights

- Tailored requirements based on RCC 319-19 rather than draft AFSPCMAN 91-712.
- Tailored coding guidelines based on AUTOSAR (update of MISR C++:2008).
- Includes new lessons-learned document:
  *Precautions when Writing Mission Rules for CASS-Based Systems*.
- Provides reference hash values (e.g., SHA256) of MDL file for use by wrapper.
  Aids in ensuring data integrity over communication channels.

## Drag-Corrected Impact Point (Drag IP)

- NASA algorithm adapted for use in CASS OR3.
- Provides a more realistic impact point estimate at the cost of processing time.
- Drag IP computed when vehicle subpoint is within an atmospheric region, otherwise vacuum IP is computed.
- Any number of atmospheric regions can defined and are separately tuneable.
- Drag IP can be disabled so vacuum IP is always used
  $\Longrightarrow$ all other OR3 features available at an OR2 computational speed.
- Uses U. S. Standard Atmosphere, 1976, U. S. Committee on Extension to the Standard Atmosphere (COESA) or each atmospheric region can have a user-defined atmospheric model.
- User-defined wind models required for each atmospheric region. Supports replacing wind model data for each atmospheric region on day of launch.
- Requires a user-defined drag model, either a MACH/$C_d$ table or subsonic and supersonic Ballistic Coefficient pair.
- Drag models can be switched based on user-defined expressions utilizing flight event conditions.

## Enhanced Computational Elements within Rules

- "Tables" element within Rules replaced with "Compute" element.
- List of "Table" elements replaced with list of "Assign" elements.

CASS OR2

```
<GenericRule id="My_Rule">
  <Tables>
    <Table idRef="TFLO_to_Chevron">
      <XVariable> TFLO </XVariable>
      <YVariable> velTotal </YVariable>
      <OutVariable> time_chevron </OutVariable>
    </Table>
  </Tables>
    .
    .
    .
```

CASS OR3

```
<GenericRule id="My_Rule">
  <Compute>
    <Assign id="time_chevron">
      TFLO_to_Chevron(TFLO, velTotal)
    </Assign>
  </Compute>
    .
    .
    .
```

- "Assign" element uses a function-call syntax.
  - Target variable for the assignment (i.e., time_chevron) is given as the "id" attribute value in the "Assign" element. In OR2 it was the content text of the "OutVariable" element.
  - The table name (i.e., TFLO_to_Chevron) is used as the name of the function to be called. In OR2 it was the value of the "idRef" attribute of the "Table" element.
  - Inputs for the table (i.e., TFLO and velTotal) appear as arguments passed to the function. In OR2 they were the content text of the "XVariable" and "YVariable" elements.

## Enhanced Computational Elements within Rules (Continued)

- Math expression of "Assign" element can be any one of the following:
  - A user-defined table with one, two, or three arguments. (OR3 supports 3-way tables in addition to 1- and 2-way tables.)
  - A literal value; integer, floating point, Boolean, or time.
  - An existing name in the database; copy value in named database entity to target.
  - A built-in function name. Available functions:
    - Absolute Value for integer or floating point arguments.
    - Equivalence for integer, floating point, or time arguments.
    - Sum for integer or floating point arguments.
    - Difference for integer, floating point arguments, or time.
    - Absolute Value for integer or floating point arguments.
    - Product for integer or floating point arguments.
    - Quotient for integer or floating point arguments.
    - Maximum for integer or floating point arguments.
    - Minimum for integer or floating point arguments.
    - Euclidean norm for 3-element integer or floating point vector arguments.
    - Great Circle distance between two point given in latitude and longitude.
    - Euclidean distance between two 3-element EFG vector arguments.
    - Angle between two 3-element EFG vector arguments.

## User-Defined Tracking Sensor Variables

- In "UserDefines" section, a constant or variable can be added to each tracking sensor's data set via the new "sensor" attribute. For example:

```
<Variable name="old_latIP" type="float" sensor="true"> LaunchIPlat </Variable>
<Variable name="old_lonIP" type="float" sensor="true"> LaunchIPlon </Variable>
<Variable name="IP_Cycle_Track" type="float" sensor="true"> 0.0 </Variable>
<Variable name="IP_Total_Track" type="float" sensor="true"> 0.0 </Variable>
```

- Tracking sensor variables can be set within a rule using the "Compute" element.

```
<Compute>
    <Assign id="IP_Total_Track"> distance(LaunchIPlat, LaunchIPlon, latIP, lonIP) </Assign>
    <Assign id="IP_Cycle_Track"> distance(old_latIP, old_lonIP, latIP, lonIP) </Assign>
    <Assign id="old_latIP"> latIP </Assign>
    <Assign id="old_lonIP"> lonIP </Assign>
</Compute>
```

- Tracking Sensor constants or variables can be referenced in template and non-template Mission Rules.

## Required User-Defined Vehicle Sensor

- Consolidates all non-tracking sensor inputs into a generalized interface.
- All vehicle sensor inputs are defined explicitly as part of the `<Vehicle>` sensor definition within the `<Sensors>` section of the Mission Rules.
- Vehicle inputs can be defined as integer, floating point, Boolean, or time.
- Vehicle sensor inputs include current system time (time value), liftoff 'A' state (Boolean value), and listfoff 'B' state (Boolean value), defined with the `<Required_Data>` element.
  - Eliminates need for `Update_Liftoff_Switches` function, which has been removed from OR3 code base.
  - Eliminates need for passing the current system time to the `Update` function. Instead, `Update` function requires a vehicle sensor object as its single argument.
- Additional vehicle inputs can be defined with the `<Additional_Data>` element.
  - Intent is to provide data such that flight events (e.g., ignition, burnout) are detected more accurately and timely.
  - Wrapper and its hardware must be able to accept external vehicle data and store data items into the vehicle sensor in the proper field.

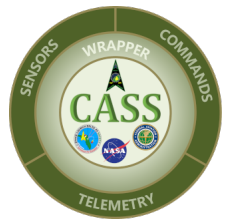
## Tracking Sensor Filters

- Tracking sensor filters check state data: measurement time, position vector, velocity vector.
- Two filters available:
  - `Valid_Representation`: Verifies state data contains valid representation of numeric data.
  - `Limit_Filter`: In addition to verifying valid representation, verifies state data is within defined limits provided by Mission Rules writer, as follows:

    measurement time $>$ `Minimum_Time`
    position magnitude $\geq$ `Minimum_Position`
    position magnitude $\leq$ `Maximum_Position`
    velocity magnitude $\leq$ `Maximum_Velocity`
- Filter result saved in Boolean flag `passedFilterTest`.
- Allows custom filtering of each tracking sensor's state data. Contrast with `realityCheck` flag that checks each tracking sensor's computed altitude, velocity magnitude, and computed total acceleration against global limits defined in the `<Settings>` section of Mission Rules.

## Removed "`persist`" Attribute for Expressions

- The "`persist`" attribute was intended to provide a filter for raw data transients, when raw tracking sensor data was used in `<ApplyWhen>` (OR1) and `<InvalidWhen>` (OR2) expressions.
- Instead, the "`persist`" attribute is often used as a general delay factor.
- Complicates rules and delays when rules have dependencies, i.e., one rule depends on the results of a previous rule.
- May incorrectly delay a response when used in `<FireWhen>` (OR1) or `<Result>` (OR2) expressions.
- Tracking sensor filter introduced in OR3 to mitigate certain types of transients. Additional filters can be added to future OR3 updates to improve transient detection.
- Removing "`persist`" attribute removes complications of rule dependencies.
- Delays for action response is more accurately handled via stair step commands.



## Dynamic Boundary Definitions and Computations

- Dynamic boundary definitions are unified into a single section of OR3 Mission Rules, `<Boundaries>` section.
  - OR2 dynamic boundary definitions used two sections of the Mission Rules, `<Tables>` and `<Boundaries>` sections; a convoluted definition.
  - OR3 method provides better visualization of the dynamic boundary; a simple definition.
  - OR3 method defines a group of static boundaries as a dynamic boundary, with a selector constant assigned to each static boundary.
- Run-time selector values are used to interpolate between static boundaries to create a dynamic boundary in real-time.
  - OR2 method forced the same global parameter to be used for all rules that referenced the dynamic boundary, regardless of the tracking sensor used by the rule.
  - OR3 method allows a global parameter to be used, to maintain OR2 capability.
  - OR3 method allows a tracking sensor variable to be used; providing a sensor-specific capability.
  - OR3 method allows a value computed from various data items; providing an adaptive capability.
- New implementation is more efficient.
  - OR2 method saves latitude and longitude for every vertex of every static boundary.
    OR3 method saves none of the vertex latitudes or longitudes; a significant memory reduction.
  - Both OR2 and OR3 depend on a unit vector for each boundary vertex.
    - OR2 selects two bounding vertices then interpolates latitude and longitude of the dynamic boundary's vertex point, then computes interpolated vertex point's unit vector.
    - OR3 selects two bounding unit vectors and interpolates the angle between them to directly compute the interpolated vertex point's unit vector.
  - OR3 implementation requires less memory and computes results faster.

## Real-Time Boundary-Dependent Green Time Rule

- Formal Green Time rule replaces "manual" green time rules.
- Template rule. Computations are per-sensor.
- When tracking sensor data is available, per sensor green time threshold is set based on reference point (i.e., either vehicle subpoint or estimated impact point) being within a green time boundary.
- Multiple green time boundaries can be defined and searched based on a user-defined sequence.
- Green time thresholds are associated with each user-defined green time boundary.
- Green time thresholds can be constants or computed values.
- When no tracking sensor data is available, per sensor *no data time* is computed from current system time and the sensor's last valid data time.
- Sensor's *no data time* is compared to sensor's green time threshold to determine if green time has elapsed.
- `CommandWhen` expression can vote to determine if a green time violation has occurred.

## Miscellaneous

- New reference frame parameter, `posElevation` (look elevation angle).
- Upgraded from 16-bit CRC to 32-bit CRC.
- Upgraded from tailored AFSPCMAN 91-712 (2004 DRAFT standard) to tailored RCC 319-19 (current RCC standard).
- Upgraded from tailored MISRA C++ to certified compliance with tailored AUTOSAR C++ coding guidelines.
- New documentation *Precautions when Writing Mission Rules for CASS-Based Systems*.