Events Interface Proposal

Introduction

MAVLink currently does not provide a mechanism to let one component inform others about sporadic events or state changes with certain delivery. The most prominent case is for an autopilot to inform a user via GCS about state changes (e.g. switching into a failsafe mode). This has traditionally been solved by sending string text messages that are then displayed directly. However this has several shortcomings:

- The message could be lost, there is no retransmission
- Translation is not possible (or at least very impractical)
- Limited text length, no URLs
- Not suited for automated analysis, or consumption by other components
- Increased flash size requirements for an autopilot binary

The goal of this document is to provide a generic solution for that.

Further background: http://discuss.px4.io/t/event-interface/2318

Scope

To simplify the design considerations, the following provides a non-exhaustive list of what is within and what is not within the scope of the events interface:
Within scope:

- (Sporadic) state changes that other systems/components might be interested in.
- The state change should be of low frequency (vehicle attitude or the current battery level would be high frequency. Whereas reaching critical battery level can be an event). High frequency state updates should use existing message streams.
- Preflight and system health are within the scope, but are corner-cases in the sense that they also need to be sent when a GCS connects.

Out of scope:

- Not meant to be a general replacement for printf.
- Not meant for messages to debug internal states.
- Information that is requested by a GCS (for example the flight duration).
- Camera trigger command. MAVLink provides command messages for that.
- Geotagging message (mostly because it has a lot of payload, but if needed could be made possible with events).

Requirements & Goals

This is a list of requirements that the interface must fulfill:

- Reliable delivery, some kind of retransmission.

- Provide a consistent interface to report system health and arming checks.
- Minimize buffer requirements on the autopilot side.
- Minimize binary message length, preferably all equal size to simplify buffer handling.
- Generic, autopilot- and GCS-agnostic.
- Must be extensible, but also long-term stable.
- Allow arguments to be attached to an event.
    - Possible types: uint8, int8, uint16, int16, uint32, int32, int64, uint64, float
    - Allow for enum's and bit fields on top of these types
- Support message translation.
- Allow for automated processing (for example from a flight log containing events).
- Minimize the amount of auto-generated code for embedded implementations.
- Target an overall events volume of <1 Hz on average. Though that should scale with increasing buffer lengths and other protocol parameters adjustments, like retransmission timeouts.
- Events can be consumed by any component, i.e. are broadcast.
- They can be targeted towards a certain component.
- Any component can send events (components running on the companion. Also a GCS, but that's not planned initially).
- Events have metadata, like a log level. They can also have a detailed, more extensive description, possibly with URLs.

Example Use-cases

This is a very short list of example usages:

- Subsystem and sensor health, arming checks (see below for details)
- Failsafe mode triggering (with trigger reason). A description should provide instructions on how to change the configuration
- Low on SD card storage
- Battery levels
- EKF reset
- System overload
- Takeoff / landing
- Replace the existing text-based calibration protocol

Protocol

This section describes the protocol for sending and receiving events, and the required MAVLink messages.
Each event has an ID, a unique name and an incremental sequence number. The sequence is allowed to wrap, so a 16-bit unsigned integer is sufficient.
A sender then increments the sequence number for each new event and sends it. A receiver keeps track of the sequence number for each component individually (that it is interested in).

In addition, a sender periodically broadcasts its current sequence number (with a suggested interval of 3 seconds to avoid spamming, but that can be adjusted as needed). Each receiver can then check that against the last received event sequence. If either a new incoming event or the current broadcast sequence does not match the expected one, it will re-request the missing events in between.
This way a sender does not need to keep track of individual receivers, but it just has to keep the last N events buffered. If a requested event already got evicted from the buffer the sender shall return an error message.
With regard to the sequence number there is a special case that needs to be handled: the sequence number might have to be reset in certain situations, for example a vehicle reboot or a component restart. A sender can indicate this with a flag in the current sequence message.

MAVLink Messages

The previously described protocol can be implemented with the following MAVLink messages: Events are sent in a generic EVENT message that contains the event ID and an array for the arguments. The arguments and types depend on the event ID. The events are defined in a separate XML file (see below).
The messages are also here.

```
      <enum name="MAV_EVENT_ERROR_REASON">
      <description>Reason for an event error response.</description>
      <entry value="0" name="MAV_EVENT_ERROR_REASON_UNAVAILABLE">
      <description>The requested event is not available
(anymore).</description>
      </entry>
      </enum>
      <enum name="MAV_EVENT_CURRENT_SEQUENCE_FLAGS">
      <description>Flags for CURRENT_EVENT_SEQUENCE.</description>
      <entry value="1" name="MAV_EVENT_CURRENT_SEQUENCE_FLAGS_RESET">
      <description>A sequence reset has happened (e.g. vehicle
reboot).</description>
      </entry>
      </enum>

      <message id="375" name="EVENT">
       <description>Most events are broadcast, some can be specific to a
target component.</description>
      <field type="uint8_t" name="destination_component">Component ID</field>
      <field type="uint8_t" name="destination_system">Component ID</field>
      <field type="uint16_t" name="id">Event ID</field>
      <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp (time
since system boot when the event happened).</field>
```

```
      <field type="uint16_t" name="sequence">Sequence number.</field>
      <field type="uint8_t[40]" name="arguments">Arguments depend on event
ID.</field>
      </message>
      <message id="376" name="CURRENT_EVENT_SEQUENCE">
      <description>Broadcast for the current latest event sequence number for
a component.</description>
      <field type="uint16_t" name="sequence">Sequence number.</field>
      <field type="uint8_t" name="flags"
enum="MAV_EVENT_CURRENT_SEQUENCE_FLAGS" display="bitmask">Flag bitset.</field>
      </message>
      <message id="377" name="REQUEST_EVENT">
      <description>Request an event to be (re-)sent.</description>
      <field type="uint8_t" name="target_system">System ID</field>
      <field type="uint8_t" name="target_component">Component ID</field>
      <field type="uint16_t" name="sequence">Sequence number of the requested
event.</field>
      </message>
      <message id="378" name="EVENT_ERROR">
      <description>Response to a REQUEST_EVENT in case of an error (e.g. the
event is not available anymore).</description>
      <field type="uint8_t" name="target_system">System ID</field>
      <field type="uint8_t" name="target_component">Component ID</field>
      <field type="uint16_t" name="sequence">Sequence number.</field>
      <field type="uint16_t" name="sequence_oldest_available">Oldest Sequence
number that is still available after the sequence set in
REQUEST_EVENT.</field>
      <field type="uint8_t" name="reason" enum="MAV_EVENT_ERROR_REASON">Error
reason.</field>
      </message>
```

Event Definitions

Similar to mavlink message definitions, events are defined in an XML file.
This can then either be read directly by a GCS or converted into an intermediate format. For other projects like the autopilot (but also useful for a GCS), code can be generated from templates (e.g. jinja).
Properties:

- Each event has a unique ID (4 bytes) and unique name
- Each event can have 2 different log levels:
    - An external log level: for everything that is sent on MAVLink.

    - An optional internal log level (defaults to the external log level): project-internal log level, e.g. for file logs.
    - Log levels definition: like MAV_SEVERITY, with these changes:
        - Protocol (7) instead of Debug: for events that should not be shown in a UI (for example the calibration protocol).
        - Disabled (8): it should not be sent (e.g. not sent via MAVLink for external, or not logged to the file for internal).
- Each event has a (short) message, preferably one line. This can contain the arguments, in the form of '{1}', similar to printf.
- Each event can have a (longer) description, meant for further information (e.g. how to resolve a problem). It might contain:
    - URLs
    - Links to parameters, like `<param>SYS_AUTOSTART</param>`.
        - A GCS should allow a user to click on it and then show the parameter change dialog.
    - Conditions based on parameters. E.g. `<show_if condition='arg1>5'>...</show_if>` or `<show_if condition='bitfield&4'>...</show_if>`.This allows to show more relevant information.
      This is a possible future extension.
- Arguments: an event can have a set of arguments, up to a maximum total of N bytes. As events generally do not require many arguments, and to reduce buffer sizes, a maximum of N=16 bytes is suggested.
  Then each message requires 4B timestamp + 4B ID + 2B sequence + 1B log levels + 25B arguments = 36 bytes, or 360 bytes to buffer the 10 latest events.
  Note that as the MAVLink EVENT message uses a higher limit, N can still be increased later on up to that limit.
- Support for Enum's and Bitfields, as MAVLink does.
- Events can be marked as deprecated (arguments of existing events cannot be changed. If required, they have to be replaced with new events).
    - Messages and descriptions of existing events can be changed, as long as the semantic stays the same.
    - Enum's and bitfield's are allowed to be extended
- Events can be grouped, for instance all health events can be in a group with name 'health'. A GCS can use this metadata for example to display a group in a separate view.

For an example XML, see here.

Subsystem and Sensor Health, Arming Checks

An important use-case for the new interface is to improve system health and arming checks reporting. This section outlines how that can be achieved.

As there is a close relationship between health and arming checks (arming checks will likely fail for an unhealthy system), they are designed together.

Goals

In order to create a good user experience, the following goals should be met:

1. Differentiate between health (vehicle conditions that require some kind of maintenance, either hardware or software fixes) and arming conditions (e.g. no GPS lock, or sensor calibration required, problems that are specific to an environment and/or can be fixed 'on the field').
   This allows a GCS to display a separate health page.
2. Drive parts of the GCS UI. This can be used for sensor inputs (e.g. show the GPS state in red/yellow/green, or indicate a missing RC), or to enable/disable buttons (e.g. 'start mission', 'arm the vehicle' or 'do a takeoff').
3. Provide detailed and specific information on how to resolve issues, or links to documentation.
4. Provide a (minimal) list of problems that need to be solved to arm in the current flight mode,
5. but also provide a list of overall problems. For example, so that a user can see why it is not possible to start a mission, independent in which flight mode the vehicle currently is.

(1) can easily be met by separating all messages between health and arming checks, although it can be expected that there are cases where it is not clear to which one a problem belongs to.
In order to meet (2), there is a summary message for health and for arming checks, with predefined fields for all major system components/sensors/flight mode categories. These can be used in a GCS for hard-wired logic to drive the UI, but they only provide high-level information (no failure explanations).
To meet (3)-(5), and to reduce the coupling/dependency between GCS and autopilot, each error/warning triggers a separate event. These can be very specific (potentially with arguments) and at the same time a GCS can treat them generically, e.g. by adding them to a list of problems. This allows you to add/change error messages without having to update GCS logic.
Specifically in QGroundControl, this might be implemented as an additional 'Problems' dropdown list (like the existing vehicle messages), and separated into 'overall' and 'current flight mode' tabs.
The drawback of having individual error events is that there might be many events sent in some cases (e.g. during vehicle setup).

Reporting the health of other components (e.g. avoidance system or camera on the companion): the autopilot should collect all of these and is responsible to report them to the GCS, since for arming checks it needs to know and evaluate the health of other on-vehicle components anyway.

Health

The health summary message contains a set of fields for each high-level sensor type or subsystem component (HEALTH_TYPE bitfield, similar to MAV_SYS_STATUS_SENSOR with some cleanup and additions):

```
<event id="10" name="HEALTH_SUMMARY" loglevel="protocol"
loglevel_internal="info">
    <message>Overall subsystem and sensor health summary message</message>
    <description></description>
    <argument name="is_present" enum="HEALTH_TYPE">Bitset for each component
if present or not</argument>
    <argument name="error" enum="HEALTH_TYPE">Bitset for each component if
there is a failure or fatal error</argument>
    <argument name="warning" enum="HEALTH_TYPE">Bitset for each component if
there are warnings</argument>
</event>
```

There are 3 bits for each subsystem/sensor:

- If the *is_present* bit is set, and the *error* bit as well, then there is a problem with that sensor. If the *is_present* bit is not set but the *error* bit is, it means that sensor is required but missing.
- The *warning* bit can be set for example if one out of several gyro's fail (and the vehicle can still fly with the remaining ones).
- If all 3 bits are 0, a UI can hide that entry, as it's not relevant.
- Even if the *is_present* and *error* bits are both set, the preflight checks might still pass (for optional sensors, or depending on the flight mode).

General:

- If the GPS has no lock, it would not be a warning or failure, since that is not a health issue.
- Similar if a sensor requires calibration: that should be handled in the arming checks message.
- Also for mission check failures or if no mission is available: that is not a health error/warning. A missing SD card could be a warning, whereas a corrupt file system might trigger a logging and mission error.
- Health errors generally mean that you cannot fly or only in limited modes (e.g. Manual), and the vehicle needs to be repaired (or the software needs to be fixed), except for optional sensors (whether a sensor is optional can be dependent on the vehicle configuration).

In addition to the summary, each error/warning will trigger one or more specific events that describe the problem in detail. They all contain a HEALTH_TYPE argument so that they can be assigned to the affected sensor/subsystem automatically. In addition, they contain a FLIGHT_MODE_CATEGORY field (see below), so that health issues can be assigned to flight mode categories (for example a broken GPS can be an issue for position control, but it is not required for Manual mode).

Note: the health events replace the health state in SYS_STATUS, as it is more flexible, provides more specific error messages (that can be associated with sensors/subsystems) and differentiates between health and arming conditions.

An autopilot shall always send the health summary message first, and then individual warnings/errors. A GCS can then clear previous messages whenever a new summary is received. This means that an individual sensor problem will trigger re-sending all health issues. The same applies to the arming checks summary message.

Arming Checks (aka Preflight Checks)

As for the health, there is an arming checks summary message:

```
<enum name="FLIGHT_MODE_CATEGORY" type="uint8_t" display="bitset">
<description>Flight mode categories</description>
<entry value="1" name="current"><description>Current flight
mode</description></entry>
<entry value="2" name="manual"><description>Fully Manual flight without any
stabilization</description></entry>
<entry value="4" name="rate"><description>Rate-stabilized flight mode
(acro)</description></entry>
<entry value="8" name="attitude"><description>Attitude-stabilized flight
mode</description></entry>
<entry value="16" name="altitude"><description>Altitude-stabilized flight
mode</description></entry>
<entry value="32" name="position"><description>Position-controlled flight
mode</description></entry>
<entry value="64" name="autonomous"><description>Autonomous flight (e.g. RTL,
Takeoff)</description></entry>
<entry value="128" name="mission"><description></description>Mission flights
(e.g. waypoints, survey)</entry>
</enum>

<event id="20" name="ARMING_CHECKS_SUMMARY" loglevel="protocol"
loglevel_internal="info">
<message>Overall arming checks status message</message>
```

```
<description></description>
<argument name="error" enum="HEALTH_TYPE">Bitset for each component if there
is an error (over all flight modes)</argument>
<argument name="warning" enum="HEALTH_TYPE">Bitset for each component if there
are warnings</argument>
<argument name="can_arm" enum="FLIGHT_MODE_CATEGORY">Bitset for each flight
modes if arming is allowed</argument>
<argument name="flags" enum="ARMING_CHECKS_SUMMARY_FLAGS"></argument>
</event>
```

There is a FLIGHT_MODE_CATEGORY bit field that should abstract all the existing flight modes to roughly the level of autonomy (which should also work for future flight modes). The idea is, the higher the level of autonomy, the more potential requirements you have (for example mission requires autonomous flight, but can have additional requirements on top. And for example if there is an error with position mode, you would have that error also for autonomous and mission modes (the design does not require this however)). The ‘current’ bit is meant for the currently active flight mode. (And as it is an abstraction, exotic flight modes such as Offboard are difficult to map).
The message contains these fields:

- *error*/*warning*: error and warning bit for each sensor/component. This can be used for example to show the GPS state in red/yellow/green: the *error* bit is set if there is no GPS fix, and the *warning* bit if the GPS accuracy is low (independent from the flight mode).
- *can_arm*: Whether the vehicle can be armed for all flight mode categories and for the current flight mode. This can be used for an ‘arm’ or a ‘start mission’ button. However, a cleared bit does not mean it is not possible to switch into certain flight modes (without arming).

As for the health, each arming error or warning bit can trigger another specific event with detailed description. Each of these also contain a field with the bitfield FLIGHT_MODE_CATEGORY, to associate the message with the affected flight modes.
In case of health errors that already triggered events, there should not be an additional arming error event message. The error bit in ARMING_CHECKS_SUMMARY should however still be set.

Other points:

- If a user still tries to arm via RC with failing arming checks, a generic error event can be sent, among the lines: ‘Arming denied, see arming checks for details’.
- Switching modes: if a mode-switch is denied, an error message with the failure reason (possibly as an enum) can be sent.
- Failsafe events also trigger separate events, but the arming checks also need to be resent, for example to indicate a GPS fix loss.

Discussion points:

- The above structure requires an autopilot to run the checks twice or store/collect the events, so that the summary can be sent first. As these checks need to be repeatable and deterministic, this should not be an issue other than the added runtime.
- The arming checks need to be run in-flight as well to reflect state changes, such as GPS fix loss. So it might be better to find another name rather than arming checks.
- The above is quite demanding on the autopilot implementation (in terms of keeping track of states, filling in messages appropriately and sending only at the time of actual changes). However it can also lead to a clean(er) implementation, as for example the *can_arm* bit for the current flight mode is the same as used internally to check for an arming transition.
  It also allows for a lot of flexibility on the GCS side in designing a UI or UX.
- Health and arming checks need to be sent when:
  - One of the states changes (e.g. sensor becomes unhealthy, GPS fix lost, …)
  - The flight mode changes (if that changes the list of problems for the current mode)
  - A GCS connects. This might not be reliably detectable, for example when multiple GCS are behind a proxy/router. Alternatives are:
    - A GCS requests the states via MAVLink command
    - Send the states upon a parameter list request (less ideal)

Limitations

Limitations of the above protocol and definitions:

- For event arguments: double's or arrays (also no strings) to keep events short, and limit the total length. It should be desirable to avoid strings altogether.
  One useful case for strings is to inform a user about the log file name. But an event could transmit only the numbers used in the file name and store the file name template with the event message (e.g. logs/{1}_{2}_{3}.log).
- A GCS cannot show events that it does not know yet, only the event ID.
- A sender does not know if/when a receiver received an event - this should not be required though.
- The design will not work if multiple components with the same component and system ID start *sending* events. That might happen if multiple GCS are connected (and send events). However, this is more of a setup/MAVLink problem, not with the protocol.

Implementation

There is a minimal end-to-end implementation for the protocol based on the PX4 Firmware and QGroundControl (without the code generation or XML parsing yet. The events are hardcoded and code that should be generated is marked. The QGroundControl side is implemented as independent library):

- https://github.com/bkueng/Firmware/commit/d2177e852bdb42f54b1d4949c869e7f18ebf7027

- https://github.com/bkueng/qgroundcontrol/commit/c56fcd37c3507606987ddae5d552b6d0b09e12c9

Open Questions

This is a list of open questions:

- What should the translation workflow look like? Can it be based on the events XML file, or use an intermediate format. If so, which one?
  A GCS should read these (or there needs to be another transformation step)
- Support different dialects, like MAVLink does?
- Should arguments have units (like meters, degrees)? This would allow it to display 2000m as 2km, or switch displayed units between metric or imperial. It would however increase the implementation complexity.