AP_GPS_JAVAD_GREIS

Technical Design and Implementation Report -Tested Revision

Field	Value
Subsystem	ArduPilot AP_GPS
Backend	AP_GPS_JAVAD_GREIS
GPS type value	GPS_TYPE_JAVAD_GREIS = 27
Protocol	Javad GREIS/JPS standard frame format

1. Executive Summary

AP_GPS_JAVAD_GREIS is a custom ArduPilot GPS backend for Javad GNSS receivers using the GREIS/JPS protocol. It allows ArduPilot to ingest binary GREIS navigation messages directly from a Javad receiver over a serial UART and publish the decoded navigation solution through the standard AP_GPS::GPS_State interface.

The backend is selectable with GPS1_TYPE=27 and integrates into the standard AP_GPS frontend, scheduler, UART, EKF/INS consumer, logging, MAVLink, and RTCM paths used by other GPS backends.

The active backend supports GT, ST, RT, ET, PG, VG, PV, SG, DP, PS, AR, RE, and ER. MF, lowercase Pg, and PgWgs are not part of the current backend. The core GNSS implementation has been successfully hardware-tested with a real Javad receiver and Pixhawk4.

2. Project Changes Compared to Clean ArduPilot

The integration adds a dedicated Javad GREIS backend and the minimal frontend/build-system changes required to expose and compile it. Stale repository commit hashes and line-number-specific references are intentionally omitted because they are not stable technical properties of the implementation.

File	Type	Purpose
libraries/AP_GPS/AP_GPS_JAVAD_GREIS.h	New	Declares backend types, parser/configuration state, message and epoch structures.
libraries/AP_GPS/AP_GPS_JAVAD_GREIS.cpp	New	Implements UART parsing, GREIS decoding, epoch assembly, receiver configuration, and GPS_State publication.
libraries/AP_GPS/AP_GPS.h	Modified	Adds GPS_TYPE_JAVAD_GREIS = 27 and backend access integration.
libraries/AP_GPS/AP_GPS.cpp	Modified	Includes and constructs the backend in the GPS detection path.
libraries/AP_GPS/AP_GPS_Params.cpp	Modified	Adds parameter metadata so GPS1_TYPE=27 appears as JavadGREIS.
libraries/AP_GPS/AP_GPS_config.h	Modified	Adds AP_GPS_JAVAD_GREIS_ENABLED feature guard.
libraries/AP_GPS/wscript	Modified	Adds the backend to double-precision AP_GPS sources.
Tools/scripts/build_options.py	Modified	Adds the optional JAVAD_GREIS GPS-driver build feature.

3. Overall Architecture

```
Javad GNSS receiver
    | GREIS/JPS UART bytes
    v
AP_HAL UARTDriver / SERIALx
    v
AP_GPS_JAVAD_GREIS
    | parser / checksum / decoder / config
    v
AP_GPS::GPS_State
    v
AP_GPS frontend
    v
EKF / INS consumers -> MAVLink -> Mission Planner / GCS
```

Layer	Responsibility
Javad GNSS receiver	Produces GREIS/JPS messages and accepts ASCII configuration commands.
UART / AP_HAL	Moves raw serial bytes between the receiver and autopilot.
AP_GPS_JAVAD_GREIS	Parses frames, validates checksums, decodes navigation data, configures output, and publishes GPS_State.
AP_GPS frontend	Owns GPS instances, timing, detection, update scheduling, and injection entry points.
EKF / INS	Consumes standardized GPS_State for vehicle navigation estimation.
MAVLink / GCS	Publishes GPS fields and accepts standard RTCM injection messages.

4. ArduPilot Integration

AP_GPS.h: Defines GPS_TYPE_JAVAD_GREIS = 27 and the backend integration points.

AP_GPS.cpp: Includes AP_GPS_JAVAD_GREIS.h, constructs the backend in the GPS detection path, and leaves receiver-specific initialization to the backend rather than the generic initialization blob.

AP_GPS_Params.cpp: Publishes 27:JavadGREIS in GPS type parameter metadata.

AP_GPS_config.h: Defines AP_GPS_JAVAD_GREIS_ENABLED using the standard AP_GPS backend feature-guard pattern.

wscript: Builds AP_GPS_JAVAD_GREIS.cpp as a double-precision AP_GPS source because the backend decodes double-precision geodetic/ECEF values.

build_options.py: Exposes the Javad backend as an optional GPS driver feature for custom builds.

5. Backend Creation

```
Power on
  -> HAL runtime
  -> vehicle setup
  -> gps.init()
  -> AP_GPS::update()
  -> AP_GPS::update_instance()
  -> AP_GPS::_detect_instance()
  -> new AP_GPS_JAVAD_GREIS(...)
  -> drivers[instance] = backend
  -> later: drivers[instance]->read()
```

The backend is created through the standard AP_GPS detection path when the configured GPS type is Javad GREIS. Once stored in the drivers[] array, later GPS update cycles invoke the backend through the AP_GPS_Backend virtual read() interface.

6. Scheduler Execution

ArduPilot schedules AP_GPS::update() periodically from the vehicle scheduler. AP_GPS::update_instance() either detects a missing backend or calls the already-created backend read() method. Virtual dispatch therefore reaches AP_GPS_JAVAD_GREIS::read() for the Javad instance.

```
AP_Scheduler
   -> AP_GPS::update()
      -> AP_GPS::update_instance()
         -> drivers[instance]->read()
            -> AP_GPS_JAVAD_GREIS::read()
```

7. UART Initialization

The backend uses the UART selected by ArduPilot for a serial port configured with GPS protocol. The serial manager opens GPS ports using the configured baud rate and GPS RX/TX buffers. AP_GPS disables hardware flow control for the GPS port. The Javad backend itself does not send a receiver baud-change command; normal AP_GPS baud probing remains responsible for detection across supported GPS baud rates.

For the tested Pixhawk4 setup, the receiver was connected on the GPS serial interface and successfully detected at the configured/probed baud rate.

8. GNSS Auto Configuration

GNSS auto-configuration is implemented in AP_GPS_JAVAD_GREIS::update_config(). When GPS_AUTO_CONFIG is disabled, the backend only parses incoming GREIS data. When auto-configuration is enabled, it sends the current hardware-tested GREIS stream configuration after the initial delay.

State	Meaning
Disabled	GPS_AUTO_CONFIG=0; parse incoming GREIS only.
Configure stream	Send the combined disable/enable GREIS command after the initial delay.
Active	Receiver streams RT, GT, ST, PG, VG, SG, DP, PS, and ET at 10 Hz; the backend assembles and publishes completed epochs.

```
Requested stream: RT + GT + ST + PG + VG + SG + DP + PS + ET
Publication gate: valid epoch + RT + ST + ET + GT + (PG+VG or valid PV fallback) + not already published
```

9. ASCII Command Sent to the Receiver

The current hardware-tested receiver configuration uses one combined command terminated by LF:

```
dm,;em,,jps/{RT,GT,ST,PG,VG,SG,DP,PS,ET}:0.1\n
```

The command disables existing periodic output and enables the required JPS/GREIS navigation stream with a 0.1 s period (10 Hz). The active stream uses uppercase PG only. MF, lowercase Pg, and PgWgs are not requested.

10. GREIS Message Support

Message	Requested	Role	Publication use	Purpose
GT	Yes	Time	Required	GPS week and time-of-week.
ST	Yes	Epoch annotation	Required	Annotates the active RT epoch; ST key must match RT.
RT / ~~	Yes	Epoch start	Required	Starts/resets the active epoch.
ET / ::	Yes	Epoch close	Required	Closes the active epoch.
PG	Yes	Primary position	PG + VG	Uppercase geodetic position; 30-byte minimum payload.
VG	Yes	Primary velocity	PG + VG	Geodetic velocity paired with PG.
PV	No	Fallback navigation	Fallback	ECEF position/velocity converted to geodetic + NED when primary data are unavailable.
SG	Yes	Accuracy	Optional	RMS position/velocity accuracy enrichment.
DP	Yes	DOP	Optional	HDOP/VDOP enrichment.
PS	Yes	Satellite/status	Optional	Satellite count and solution enrichment.
AR	No	Stored only	Optional	Rotation angles retained but not published as heading.
RE / ER	Not streamed; command replies only	Configuration feedback	Optional	Receiver normal/error replies used by configuration logic.

11. Parser Architecture

```
UART bytes
 -> read()
 -> parse(byte): ID1 -> ID2 -> LEN1 -> LEN2 -> LEN3 -> BODY
 -> checksum validation
 -> dispatch_message()
 -> decode_*()
 -> Epoch_Data
 -> publish_epoch()
 -> AP_GPS::GPS_State
```

- CR/LF is ignored only while waiting for the first message-ID byte.
- Message ID is two printable ASCII characters.
- Length is three uppercase hexadecimal ASCII characters.
- Maximum payload length is 0x0FFF.
- Unknown messages are skipped using the advertised frame length.
- Known messages may be longer than the minimum expected layout; excess fields are ignored.
- Known messages shorter than their minimum payload are rejected.
- The stored payload buffer is bounded at 128 bytes.
- dispatch_message() attempts publication only after a decoder reports a successfully decoded message.

12. Epoch Synchronization

Message	Function	Role
RT / raw ID ~~	decode_rt()	Only epoch starter. Resets the previous epoch, marks RT present, stores the RT time key, and attaches pending GT when available.
ST	decode_st()	Annotates an active RT epoch only; accepted only when its time key matches the active RT key.
ET / raw ID ::	decode_et()	Closes the active RT epoch; never starts an epoch.

GT	decode_gt()	Provides GPS week/time-of-week and may be held pending until an RT-started epoch exists.

The epoch model is RT-driven. RT starts/resets the epoch, ST annotates the active epoch, and ET closes it. Navigation messages are accepted only for the active open epoch. Publication requires a valid, unpublished epoch with RT, ST, ET, GT, and navigation data. Navigation is complete when PG + VG is available or when a valid PV fallback has been converted. An epoch is published at most once.

13. Navigation Logic

```
Primary path:
    PG + VG
       -> GPS_State position + velocity

Fallback path:
    PV
       -> ECEF position -> latitude / longitude / height
       -> ECEF velocity -> NED velocity
       -> GPS_State position + velocity
```

Uppercase PG + VG is the primary navigation path. PV is used only when primary navigation data are unavailable and PV conversion has succeeded; PV cannot override a complete PG + VG solution.

PG provides latitude, longitude, ellipsoidal height, position sigma, and solution type. VG provides geodetic velocity plus velocity sigma and solution type. PV provides ECEF position and velocity with associated solution/accuracy fields and is converted before publication.

14. GPS_State Population

GPS_State field	Source / behavior
status	Most conservative AP_GPS_FixType among available solution types.
time_week	GT week.
time_week_ms	GT time-of-week in milliseconds.
location.lat	Radians converted to degrees * 1e7 from PG or PV.
location.lng	Radians converted to degrees * 1e7 from PG or PV.
location.alt	Ellipsoidal height in meters * 100; no AMSL conversion.
have_undulation / undulation	false / 0.0f.
velocity	NED velocity from VG or converted PV.
have_vertical_velocity	true for published navigation solution.
ground_speed / ground_course	Computed from the NED velocity.
num_sats	PS satellite count, otherwise 0.
rtk_num_sats / rtk_age_ms	Epoch satellite count / 0.
hdop / vdop	DP scaled by 100, otherwise GPS_UNKNOWN_DOP.
horizontal_accuracy	SG hpos, or PG/PV position-sigma fallback as implemented.
vertical_accuracy	SG vpos only.
speed_accuracy	SG hvel, or velocity-sigma fallback.

Fix type mapping follows the backend solution-type mapping: 0=NONE, 1=FIX_3D, 2=DGPS, 3=RTK_FLOAT, 4=RTK_FIXED, 5=STATIC, other=NONE.

15. RTCM Support

```
Mission Planner / GCS
  -> MAVLink GPS_RTCM_DATA or GPS_INJECT_DATA
  -> AP::gps().handle_msg()
  -> AP_GPS RTCM reassembly/injection
  -> drivers[instance]->inject_data()
  -> AP_GPS_Backend::inject_data()
  -> UART write to Javad receiver
```

AP_GPS_JAVAD_GREIS does not override inject_data(); it inherits the standard AP_GPS_Backend implementation, which writes raw correction bytes to the backend UART when TX space is available. Mission Planner changes are therefore not required for basic standard RTCM forwarding.

The core GNSS backend has been hardware-tested with Javad + Pixhawk4. Javad-specific RTCM correction handling and simultaneous same-UART RTCM/GREIS operation remain separate validation items unless they have been explicitly tested in the target deployment.

16. Runtime Sequence

```
Pixhawk power on
 -> vehicle/HAL setup
 -> serial_manager.init()
 -> gps.init()
 -> AP_GPS::update() / update_instance()
 -> detect and construct AP_GPS_JAVAD_GREIS
 -> later scheduler pass: drivers[instance]->read()
    -> update_config()
       -> send_command(dm,;em,,jps/{RT,GT,ST,PG,VG,SG,DP,PS,ET}:0.1\n)
    -> read UART bytes
    -> parse() -> checksum -> dispatch_message() -> decode_*()
    -> Epoch_Data -> publish_epoch() -> GPS_State -> EKF -> MAVLink -> Mission Planner
```

17. Known Limitations

Limitation	Explanation
Ellipsoidal altitude	PG altitude is treated as ellipsoidal height. The backend writes it to GPS_State.location.alt without AMSL conversion and sets have_undulation=false.
Strict epoch dependency	Publication requires the intended RT-started sequence: RT, matching ST, GT, navigation data, and ET. Missing or mis-associated markers prevent publication by design.
AR not published	AR is decoded/stored but is intentionally not written to GPS_State as heading in the current backend.
Receiver command assumptions	The backend assumes the combined dm/em syntax and requested RT,GT,ST,PG,VG,SG,DP,PS,ET stream are accepted on the connected receiver port.
RTCM validation scope	Standard AP_GPS RTCM injection is inherited. Javad-specific correction handling and same-UART RTCM behavior should be considered validated only after dedicated hardware testing.

18. Conclusion

AP_GPS_JAVAD_GREIS provides a dedicated ArduPilot GPS backend for Javad GNSS receivers using the GREIS/JPS protocol. The backend is integrated into the standard AP_GPS architecture as GPS_TYPE_JAVAD_GREIS = 27 and communicates with the receiver through a serial UART.

The implementation provides GREIS frame parsing and checksum validation, receiver auto-configuration, RT-based epoch synchronization, navigation data decoding, and publication through the standard AP_GPS::GPS_State interface. The receiver is configured with the combined 10 Hz stream command:

```
dm,;em,,jps/{RT,GT,ST,PG,VG,SG,DP,PS,ET}:0.1\n
```

The current navigation implementation uses uppercase PG + VG as the primary navigation source, while PV is retained as a fallback navigation source. Epoch synchronization is RT-driven: RT starts/resets the epoch, ST annotates the active epoch, and ET closes it. GT provides GPS timing information, while SG, DP, and PS provide optional accuracy, DOP, and satellite/status information. MF, lowercase Pg, and PgWgs are not part of the current backend.

The backend remains compatible with the standard ArduPilot RTCM injection path through the inherited AP_GPS_Backend::inject_data() mechanism. The core GNSS implementation has been successfully hardware-tested with a real Javad receiver and Pixhawk4, confirming operation of the implemented GREIS navigation path on the target hardware.

Source Reference Summary

Topic	Source reference
GPS type enum	libraries/AP_GPS/AP_GPS.h - GPS_TYPE_JAVAD_GREIS = 27
GPS UART discovery / backend construction	libraries/AP_GPS/AP_GPS.cpp
Backend parser / decoders	libraries/AP_GPS/AP_GPS_JAVAD_GREIS.cpp - parse(), message_info(), dispatch_message(), decode_*()
Epoch / publication logic	libraries/AP_GPS/AP_GPS_JAVAD_GREIS.cpp - decode_rt(), decode_st(), decode_et(), epoch_accepts_data(), epoch_complete(), publish_epoch()
Configuration	libraries/AP_GPS/AP_GPS_JAVAD_GREIS.cpp - update_config(), send_command()
RTCM injection	libraries/AP_GPS/AP_GPS.cpp and libraries/AP_GPS/GPS_Backend.cpp - standard injection path