API Definition Maintenance Guide

Single Source of Truth

All INAV JavaScript API definitions are centralized in:

```
tabs/transpiler/api/definitions/
```

When adding new INAV features, you only need to edit files in this directory.

Directory Structure

```
tabs/transpiler/api/definitions/
├── index.js          # Exports all definitions
├── flight.js         # Flight parameters (read-only)
├── override.js       # Override settings (writable)
├── rc.js             # RC channels (read-only)
├── time.js           # Time utilities
├── waypoint.js       # Waypoint navigation
└── [future additions]
```

Definition Format

Each definition file exports objects following this structure:

```
javascript
```

```
module.exports = {
  propertyName: {
    type: 'number' | 'boolean' | 'string' | 'object' | 'function',
    desc: 'Human-readable description',
    unit: 'Unit of measurement (optional)',
    readonly: true | false,
    range: [min, max], // Optional: valid value range
    inavOperand: {
      type: 2,    // OPERAND_TYPE constant
      value: 1    // Operand value for INAV
    },
    inavOperation: 27, // Optional: OPERATION constant
  },

  // Nested objects
  nestedObject: {
    type: 'object',
    desc: 'Description',
    properties: {
      subProperty: {
        type: 'number',
        desc: 'Sub-property description',
        // ... same structure as above
      }
    }
  }
};
```

What Uses These Definitions

1. Semantic Analyzer (`analyzer.js`)

- Validates property access
- Checks writable properties
- Validates value ranges
- **Auto-updates when definitions change**

2. Type Definitions (`types.js`)

- Generates TypeScript definitions for Monaco Editor
- Provides IntelliSense autocomplete

- **Auto-generates from definitions**

3. Code Generator (`codegen.js`)

- Maps JavaScript to INAV operands
- Uses `inavOperand` and `inavOperation` fields
- **Requires manual update for new operations**

4. Decompiler (`decompiler.js`)

- Reverse maps INAV to JavaScript
- Uses operand mappings from API definitions
- **Auto-updates when definitions change**

Adding a New Property

Example: Adding `flight.compassHeading`

1. Edit `tabs/transpiler/api/definitions/flight.js`:

```javascript
module.exports = {
  // ... existing properties ...

  compassHeading: {
    type: 'number',
    unit: '°',
    desc: 'Compass heading in degrees (0-359)',
    readonly: true,
    range: [0, 359],
    inavOperand: {
      type: 2,    // OPERAND_TYPE.FLIGHT
      value: 40   // FLIGHT_PARAM.COMPASS_HEADING
    }
  }
};
```

2. Update `inav_constants.js` (if needed):

Only if adding a completely new INAV firmware feature:

```javascript
```

```
const FLIGHT_PARAM = {
  // ... existing params ...
  COMPASS_HEADING: 40
};

const FLIGHT_PARAM_NAMES = {
  // ... existing names ...
  [FLIGHT_PARAM.COMPASS_HEADING]: 'compassHeading'
};
```

3. That's it!

The following automatically update:

- ✅ Semantic analyzer validates `flight.compassHeading`
- ✅ TypeScript definitions show in autocomplete
- ✅ Range checking works automatically
- ✅ Decompiler recognizes the property

Adding a New Writable Property

Example: Adding `override.vtx.frequency`

1. Edit `tabs/transpiler/api/definitions/override.js`:

```javascript
```

```
module.exports = {
  // ... existing properties ...

  vtx: {
    type: 'object',
    desc: 'VTX control',
    properties: {
      power: { /* ... */ },
      band: { /* ... */ },
      channel: { /* ... */ },

      // NEW PROPERTY
      frequency: {
        type: 'number',
        unit: 'MHz',
        desc: 'VTX frequency in MHz',
        readonly: false,  // Writable!
        range: [5000, 6000],
        inavOperation: 50  // New operation code
      }
    }
  }
};
```

2. Update `inav_constants.js`:

javascript

```
const OPERATION = {
  // ... existing operations ...
  OVERRIDE_VTX_FREQUENCY: 50
};
```

3. Update `codegen.js` (manual):

Add code generation logic:

javascript

```
// In generateAction() method
if (stmt.target === 'override.vtx.frequency') {
  return this.createLogicCondition(
    activatorId,
    OPERATION.OVERRIDE_VTX_FREQUENCY,
    OPERAND_TYPE.VALUE,
    0,
    this.valueOperand(stmt.value)
  );
}
```

Adding a New Top-Level API Object

Example: Adding `inav.sensors`

1. Create `tabs/transpiler/api/definitions/sensors.js`:

```
javascript
```

```
'use strict';

module.exports = {
  acc: {
    type: 'boolean',
    desc: 'Accelerometer sensor detected',
    readonly: true,
    inavOperand: {
      type: 2,  // FLIGHT
      value: 50 // New param ID
    }
  },

  mag: {
    type: 'boolean',
    desc: 'Magnetometer sensor detected',
    readonly: true,
    inavOperand: {
      type: 2,
      value: 51
    }
  }

  // ... more sensors
};
```

2. Update `tabs/transpiler/api/definitions/index.js`:

```
javascript

'use strict';

module.exports = {
  flight: require('./flight.js'),
  override: require('./override.js'),
  rc: require('./rc.js'),
  time: require('./time.js'),
  waypoint: require('./waypoint.js'),
  sensors: require('./sensors.js')  // ADD THIS
};
```

3. Update TypeScript types in `types.js` generation:

The type generator should automatically pick it up, but verify:

```javascript
// In generateTypeDefinitions()
dts += generateInterfaceFromDefinition('sensors', apiDefinitions.sensors);
```

Validation Checklist

When adding/modifying API definitions:

- [ ] Property has correct `type`
- [ ] Has descriptive `desc`
- [ ] Has `unit` if applicable
- [ ] `readonly` flag is correct
- [ ] `range` is specified for numeric values
- [ ] `inavOperand` maps to correct INAV constant
- [ ] `inavOperation` specified for writable properties
- [ ] Updated `index.js` if new file
- [ ] Updated `inav_constants.js` if new INAV feature
- [ ] Updated `codegen.js` for new writable properties
- [ ] Tested with sample code
- [ ] TypeScript definitions generate correctly

Testing Changes

After modifying definitions:

```javascript

```

```
// 1. Test semantic analysis
const code = `
const { sensors } = inav;
when(() => sensors.acc, () => {
  // ...
});
`;

const transpiler = new Transpiler();
const result = transpiler.transpile(code);
// Should not have errors

// 2. Test type generation
const { generateTypeDefinitions } = require('./api/types.js');
const dts = generateTypeDefinitions(apiDefinitions);
// Should include new properties

// 3. Test in Monaco Editor
// Open configurator, verify autocomplete shows new properties
```

Common Mistakes

❌ Wrong: Editing analyzer.js directly

javascript

```
// DON'T DO THIS in analyzer.js:
this.inavAPI = {
  'flight': {
    properties: ['homeDistance', 'newProperty']  // Hard-coded!
  }
};
```

✅ Right: Edit definition file

javascript

```
// DO THIS in flight.js:
module.exports = {
  newProperty: {
    type: 'number',
    desc: 'New property',
    // ...
  }
};
```

❌ Wrong: Duplicating definitions

javascript

```
// DON'T duplicate in multiple files
// decompiler.js - NO!
const FLIGHT_PARAMS = {
  1: 'homeDistance'
};

// analyzer.js - NO!
properties: ['homeDistance']
```

✅ Right: Use centralized definitions

javascript

```
// DO THIS - import from definitions
const apiDefinitions = require('./../api/definitions/index.js');
const flightDef = apiDefinitions.flight;
```

File Dependencies

```
tabs/transpiler/api/definitions/
  ├── index.js
  ├── flight.js
  ├── override.js
  └── ...
      ↓
    Used by:
     ├── analyzer.js (validation)
     ├── types.js (TypeScript generation)
```

```
├── codegen.js (code generation)
└── decompiler.js (via inav_constants.js)
```

Migration from Hardcoded Values

If you find hardcoded API definitions elsewhere in the code:

1. **Identify the hardcoded values**
2. **Check if they exist in `api/definitions/`**
3. **If not, add them to appropriate definition file**
4. **Replace hardcoded values with imports**
5. **Test thoroughly**
6. **Remove old hardcoded definitions**

Example:

javascript

```javascript
// Before (hardcoded in analyzer.js)
this.inavAPI = {
  'flight': {
    properties: ['homeDistance', 'altitude']
  }
};

// After (using definitions)
const apiDefinitions = require('./../api/definitions/index.js');
this.inavAPI = this.buildAPIStructure(apiDefinitions);
```

Summary

One Rule: Edit only `tabs/transpiler/api/definitions/*.js`

Everything else updates automatically (except `codegen.js` which requires manual updates for new operations).

This ensures:

- ✅ Single source of truth
- ✅ No duplication
- ✅ Easy maintenance

- ✅ Fewer bugs
- ✅ Automatic validation
- ✅ Automatic type generation