Development Process

Development Workflow

1. Introduction

This document outlines an agile development workflow for Stowood's sleep diagnostics medical device that integrates with the existing Quality Management System (QMS). The workflow is designed to provide flexibility and efficiency while ensuring regulatory compliance with medical device standards, particularly IEC 62304:2006+AMD1:2015 for medical device software lifecycle processes.

1.1 Purpose

To establish a structured yet flexible firmware development process that:

- Enables iterative development through 2-week sprints
- Maintains compliance with medical device regulations
- Integrates seamlessly with Stowood's existing QMS
- Ensures proper documentation for regulatory submission
- Facilitates verification and validation activities

1.2 Scope

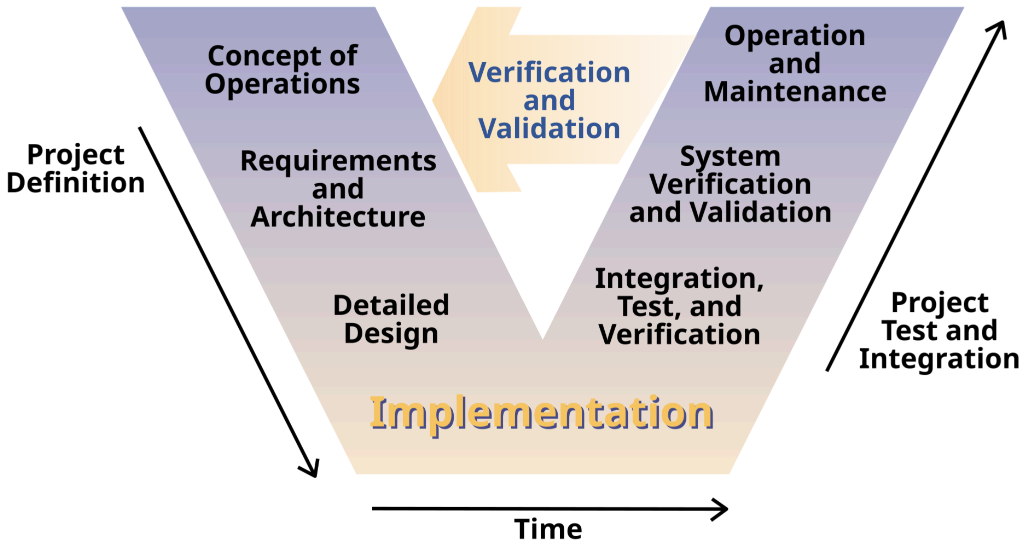
This workflow applies to the development process of Stowood's sleep diagnostics medical device. The main focus is on the hardware and firmware, with integration to backend services and proprietary software.

2. Agile Development Framework

2.1 Development Approach Overview

The firmware development will follow an agile approach with 2-week sprints while adhering to the V-model principles. The V-model is a systems development lifecycle model that pairs development activities with corresponding verification activities, ensuring each development phase has an associated testing phase. This hybrid approach allows for:

- Incremental development with regular feedback
- Risk management throughout the development process
- Traceable requirements implementation
- Continuous verification and validation activities
- Clear documentation that supports regulatory compliance


Project Definition
Concept of Operations
Requirements and Architecture
Detailed Design
Verification and Validation
Operation and Maintenance
System Verification and Validation
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

2.2 Sprint Structure

Each 2-week sprint will have the following key meetings, initially combined into one to reduce the overhead:

1. **Sprint Planning** (beginning of sprint)
    - Review and refine product backlog
    - Select requirements for implementation
    - Break down requirements into tasks
    - Commit to sprint goals
2. **Sprint Review** (end of sprint)
    - Demonstrate completed work
    - Collect feedback
    - Update progress on requirements
    - Review and plan for next sprint

The team will self-organize to determine how to best accomplish the work during the sprint period.

2.3 Roles and Responsibilities

The engineering team is self-organizing and consists of:

- **Hardware Developer**: Responsible for hardware design and integration
- **Firmware Developer**: Responsible for firmware implementation
- **Software Developer**: Responsible for application software
- **Quality Assurance**: Responsible for testing and regulatory compliance

This team structure needs to align with the QMS responsibilities while allowing for self-organisation.

Key QMS-defined roles (SSI-SOP-20) that must be fulfilled:

- **Project Lead**: Designated team member (TBD) who ensures project oversight
- **Software Manager**: Team member responsible for software/firmware development, defining requirements and risk assessment
- **QA and Regulatory Compliance Manager:** ensure that procedures are followed and that regulatory compliance is achieved
- **Support Engineer:** Reports on SSI-QF-10P any issues or observations noted by first-line support users of Stowood products
- **Independent Reviewer**: Person not directly involved in development who provides objective assessment

3. QMS Integration

3.1 Mapping Agile Process to QMS Documentation

The table below maps the agile development activities to the corresponding QMS documents:

Agile Activity	QMS Documentation	Purpose
Project Initiation	**SSI-QF-20A** (Software Safety Classification) **SSI-QF-20B** (Software Development Plan)	Establish safety classification and development approach

Requirements Gathering	**SSI-QF-20P** (Software Requirements Specification) **SSI-QF-20Q** (Software Functional Specification) **SSI-QF-20R** (Software User Requirements)	Document detailed requirements
Sprint Planning	**SSI-QF-20C** (Software Requirements Traceability)	Map requirements to sprints, ensure coverage
Development	**SSI-QF-20D** (Software Architecture Design) **SSI-QF-20N** (Software or Firmware code)	Document design and implementation
Testing	**SSI-QF-20E** (Software Test Protocol) **SSI-QF-20F** (Software Test Report)	Document test plans and results
Release	**SSI-QF-20J** (Software Release Version Control) **SSI-QF-20L** (Software Build Numbers Summary of Changes) **SSI-QF-20M** (Firmware Programmable Part Release)	Document release information
Maintenance	**SSI-QF-20G** (Software Maintenance Plan) **SSI-QF-20H** (Software Maintenance Report) **SSI-QF-10P** (Problem Report)	Document maintenance activities and issues

3.2 Design Reviews and QMS Checkpoints

To maintain compliance with the QMS while following an agile approach, formal design reviews will be conducted at key milestones:

1. **Initial Planning Review** (before sprint work begins)
    - Review safety classification (SSI-QF-20A)
    - Approve Software Development Plan (SSI-QF-20B)
    - Review requirements (SSI-QF-20P, SSI-QF-20Q, SSI-QF-20R)
2. **Architecture Review** (after initial design sprints)
    - Review Software Architecture Design (SSI-QF-20D)

    - Update Requirements Traceability Matrix (SSI-QF-20C)
    - Conduct formal design review (SSI-QF-10C)
3. **Incremental Design Reviews** (every 4-6 sprints)
    - Review implementation progress
    - Update traceability matrix
    - Assess risk management activities
    - Use SSI-QF-10C to document the design review
4. **Pre-Release Review** (before final validation)
    - Verify all requirements met
    - Ensure all tests completed
    - Complete Software Test Reports (SSI-QF-20F)
    - Prepare release documentation

Each design review will include the required participants as specified in SSI-SOP-20.

4. Agile Requirements Management

4.1 Requirements Hierarchy

Requirements will be organized in a hierarchical structure:

1. **User Requirements** (SSI-QF-20R)
    - High-level user needs and intended use requirements
    - Regulatory requirements
2. **Software Requirements** (SSI-QF-20P)
    - Functional requirements
    - Performance requirements
    - Security requirements
    - Interface requirements
3. **Functional Specifications** (SSI-QF-20Q)
    - Detailed specifications for implementation
    - Testable criteria
4. **Sprint Backlog Items**
    - Individual work items derived from requirements
    - Sized to fit within sprint timeframes
    - Linked back to formal requirements

4.2 Requirements Traceability

The Software Requirements Traceability Matrix (SSI-QF-20C) will be maintained throughout development to ensure:

- Each requirement is implemented
- Each requirement has associated verification tests
- Changes to requirements are tracked
- Requirements are traceable to risk controls where applicable

During sprint planning, the team will identify which requirements will be addressed in the upcoming sprint and update the traceability matrix accordingly.

4.3 Change Management

When requirements change during development:

1. Document changes using SSI-QF-10G (Design Change Record)
2. Assess impact on existing work and planned sprints
3. Update the Requirements Traceability Matrix
4. Re-evaluate risk assessment if needed
5. Obtain appropriate approvals per the QMS
6. Incorporate changes into sprint planning

5. Sprint Planning and Execution

5.1 Sprint Planning Process

- Backlog Refinement:
    - Break down requirements into sprint-sized tasks
    - Prioritize based on dependencies and risk
    - Define acceptance criteria for each task
    - Map tasks to SSI-QF-20C (Requirements Traceability)
- Sprint Planning:
    - Select tasks for the sprint
    - Define testing approach for each task
    - Identify dependencies and risks
    - Document sprint goals and tasks

5.2 Sprint Execution

During each sprint, developers will:

1. Implement assigned tasks
2. Implement unit testing based on a Test Driven Development (TDD) approach
3. Document code with appropriate comments, as well as repository internal documentation and wiki system
4. Participate in code reviews
5. Update development documentation
6. Track progress on tasks

5.3 Sprint Review

- Demonstrate completed functionality
- Verify acceptance criteria
- Update traceability matrix
- Document partial verification results where applicable
- Review what went well and what could be improved
- Identify process improvements
- Update risk assessment if needed
- Plan adaptations for the next sprint

5.4 Documentation Updates

During each sprint, the team will incrementally update:

- Design documentation
- Test documentation
- Risk analysis
- Traceability matrix

This ensures documentation remains current throughout development rather than being completed at the end.

6. Verification and Validation Approach

6.1 Continuous Verification

Verification activities will be integrated throughout the sprints:

1. **Unit Testing**
   - Each developer writes unit tests for their code
   - Tests are automated where possible
   - Test results are documented incrementally
2. **Code Reviews**
   - All code must be reviewed by at least one other developer
   - Reviews focus on functionality, safety, and compliance with coding standards
   - Review results are documented
3. **Static Analysis**
   - Automated static analysis tools will be run on code
   - Results reviewed and issues addressed
   - Analysis reports retained as verification evidence

6.2 Sprint-Level Integration Testing

At the end of each sprint:

1. Integrate completed components
2. Perform integration tests on the integrated components
3. Document test results using SSI-QF-20E/F templates or reference them
4. Link tests back to requirements in the traceability matrix

6.3 Milestone Verification

After several sprints (typically forming a release candidate):

1. Perform comprehensive verification testing
2. Complete formal SSI-QF-20E (Software Test Protocol)
3. Document results in SSI-QF-20F (Software Test Report)
4. Conduct a formal design review per SSI-QF-10C

6.4 Final Validation

Final validation will be conducted according to SSI-SOP-10 and will include:

1. Complete system testing
2. Validation against user requirements
3. Security testing
4. Performance testing
5. Documentation of results using QMS forms

7. Risk Management Integration

7.1 Sprint-Level Risk Assessment

During each sprint:

1. Identify potential new hazards from sprint activities
2. Update risk analysis for modified components
3. Ensure risk controls are implemented and verified
4. Document in accordance with SSI-SOP-13

7.2 Risk Traceability

Ensure that:

1. Risk controls are linked to requirements
2. Risk controls are verified through testing
3. Changes to requirements are assessed for risk impact
4. Risk information is included in design reviews

8. Release Management

8.1 Release Criteria

A firmware release (build) must meet these criteria:

1. All planned requirements for the release are implemented
2. All tests have passed or deviations are documented and justified
3. Risk assessment is complete and risks are acceptable
4. Required design reviews are complete
5. Documentation is complete and accurate

8.2 Release Documentation

For each firmware release:

1. Update SSI-QF-20J (Software Release Version Control)
2. Complete SSI-QF-20L (Software Build Numbers Summary of Changes)
3. Complete SSI-QF-20M (Firmware Programmable Part Release) if applicable
4. Ensure SSI-QF-20F (Software Test Report) is complete

8.3 Release Approval

Final approval requires:

1. Design review meeting with required participants
2. Verification that all release criteria are met
3. Formal approval documented in QMS

9. Suggested QMS Adaptations

The following adaptations to the existing QMS are recommended to better support the agile development approach:

9.1 Documentation Recommendations

1. **Incremental Documentation Updates**
    - Allow partial completion of QMS documents during development
    - Define "ready for review" states for documents
2. **Risk Documentation**
    - Implement a live risk register that can be updated throughout sprints
    - Incorporate risk reviews into sprint review

9.2 Review Process Recommendations

1. **Incremental Reviews**
    - Allow for targeted reviews of specific components
    - Implement progressive design reviews throughout development
    - Document partial verification activities during development
2. **Remote/Asynchronous Reviews**
    - Enable remote participation in reviews

    - Allow asynchronous review of code and documentation
    - Ensure review participants have sufficient time to prepare

9.3 Traceability Recommendations

1. **Digital Traceability Tools**
    - Consider implementing digital tools for traceability
    - Link sprint tasks directly to requirements
    - Automate generation of traceability reports
2. **Verification Evidence**
    - Define standards for acceptable verification evidence
    - Allow for automated test results as verification evidence
    - Establish templates for capturing verification during sprints

Practical Developer Guide

Practical Developer Guide

Introduction

This guide provides practical, day-to-day guidance for the development team. It focuses on how to integrate agile development practices with the existing Quality Management System (QMS) requirements.

Source Code Management

We use git with github for version control and continuous integration.

Git Branching Strategy

This project uses OneFlow explained below.

Core Concept

- Simpler alternative to GitFlow while maintaining the same power and flexibility
- Maintain a single eternal main branch
- Simplifies versioning and daily operations
- Results in a cleaner and more readable project history

Branch Types and Workflow

All branches should follow these naming patterns, adding the youtrack task ID.

Feature Branches

Feature branches are the primary workspace for daily development activities. They branch from and merge back into the main branch, following these key practices:

- Named using the pattern `feature/<Task-ID>-my-feature`
- Short-lived and focused on specific features

Bug Branches

Bug branches are used for fixing identified issues in the codebase:

- Named using the pattern `bug/<Bug-ID>-my-bug-fix`
- Similar to feature branches but specifically for bug fixes
- Higher priority than feature branches
- May require backporting to older versions via hotfix branches if critical

Maintenance Branches

Maintenance branches are used for repository housekeeping and non-code changes:

- Named using the pattern `misc/<Task-ID>-misc-task`
- Used for tasks like:
    - Adding or updating development scripts
    - Updating documentation
    - Modifying CI/CD configurations
    - Updating development dependencies
- Should not impact the production code
- Can be merged with less stringent review requirements

Release Branches

Release branches facilitate the preparation of software for production release:

- Named as `release/<version-number>`
- Created from specific commits on the main branch that contain all intended features
- Used for version bumps, final testing, and quality assurance
- Culminate in version tagging before being merged back to the main branch
- Allow continued development on the main branch during release preparation

Hotfix Branches

Hotfix branches handle urgent production fixes:

- Named as `hotfix/<version-number>`
- branch based on tagged release commit to be fixed
- Merge back into both the main branch
- Enable emergency fixes without disrupting ongoing development

Merge Workflow

To maintain a clean and semi-linear history, we follow these merge requirements:

Prerequisites

- All changes to the main branch must go through a Pull Request (PR) in GitHub
- Each PR requires at least one review and approval
- The process follows the `rebase + merge --no-ff` strategy

Process

1. **Prepare Your Branch**
    - Rebase onto latest main: `git checkout <branch-name> && git fetch origin && git rebase origin/main`
    - Resolve any conflicts and test build locally
    - Push your updated branch: `git push origin <branch-name> --force-with-lease`
2. **Create a Pull Request**
    - Create a new PR on GitHub by selecting your branch
    - Add a descriptive title and detailed description of changes
3. **Request and Address Reviews**
    - Select appropriate reviewers from the team
    - Wait for the review and then address feedback by making requested changes
    - Push updates to the same branch
    - Mark comments as resolved once addressed
    - Request re-review if needed
4. **Final Checks and Merge**
    - Ensure all tests/CI checks are passing
    - Verify the branch is up to date with `main`
    - Once approved, click "Merge pull request"
    - Select "Create a merge commit" option (enforces `--no-ff`)

Semi-Linear Git History

A semi-linear Git history combines the benefits of both linear and branched development:

- **Preserves feature branches** while keeping history clean and readable
- **Created using `--no-ff` merges** (no fast-forward) which force a merge commit
- **Combined with rebasing** Rebase branches onto the latest main before merging

Visual Representation:

```
Unset

* (main) Merge branch 'feature-B' |\ | * Feature B work |/ *
Merge branch 'feature-A' |\ | * Feature A work |/ * Earlier work
```

Benefits:

1. Documents feature development clearly
2. Makes project history easier to understand
3. Simplifies debugging and bisecting
4. Enables cleaner reverts when necessary

This approach balances good change tracking with history readability.

Branch Protection Rules for main

- Pull request required for all changes
- At least one approval is required
- Branch must be up to date with main
- Status checks must pass
- Force pushes and branch deletion restricted

Daily Development Workflow

During Sprint Planning

1. **Requirement Breakdown**
    - Break down requirements into manageable tasks
    - Ensure each task has clear acceptance criteria/definition of done
    - Link tasks to formal requirements in SSI-QF-20C
2. **Task Selection**
    - Select tasks for the sprint based on priority and dependencies
    - Ensure tasks have appropriate test approaches defined
    - Assign tasks based on team member expertise

During Development

1. **Before Starting a New Task**
    - Ensure the requirement is understood
    - Check for dependencies on other tasks

    - Plan your approach and testing strategy
    - Create feature branch from develop
2. **While Implementing**
    - Write code according to the coding standards
    - Create unit tests for your code
    - Document design decisions in code comments
    - Maintain traceability to requirements
3. **Before Code Review**
    - Run all unit tests
    - Perform static code analysis
    - Update relevant documentation
    - Self-review your code
4. **During Code Review**
    - Address feedback promptly
    - Document significant changes
    - Ensure requirements are still met
    - Update tests as needed
5. **After Task Completion**
    - Update the Requirements Traceability Matrix (SSI-QF-20C)
    - Document any deviations or issues
    - Merge to develop branch after approval
    - Update task status in project management tool

During Testing

1. **Unit Testing**
    - Write unit tests for all new code
    - Automate tests where possible
    - Document test results for future reference
2. **Integration Testing**
    - Test integration with other components
    - Document integration issues in SSI-QF-10P (Problem Report)
    - Verify requirements are met

During Sprint Review

1. **Preparation**
    - Prepare demonstration of completed work
    - Update Requirements Traceability Matrix (SSI-QF-20C)
    - Document any outstanding issues
2. **During Review**
    - Demonstrate working functionality

- Discuss any issues or challenges
- Capture feedback for future sprints

Managing Documentation During Development

Incremental Documentation Approach

Rather than completing documents all at once, update them incrementally throughout development:

1. **During Planning**
   - Initial versions of requirements documents
   - Software Development Plan updates
2. **During Implementation**
   - Architecture updates
   - Traceability matrix updates
   - Code documentation
3. **During Testing**
   - Test protocols and reports
   - Verification evidence

QMS Documents Quick Reference

When you need to…	**Update these documents**
Start a new feature	• Requirements Traceability (SSI-QF-20C) • Functional Specification (SSI-QF-20Q) if needed
Change design	• Software Architecture (SSI-QF-20D) • Design Change Record (SSI-QF-10G)
Fix a bug	• Problem Report (SSI-QF-10P) • Update changelog
Test functionality	• Test Protocol (SSI-QF-20E) • Test Report (SSI-QF-20F)
Prepare for release	• Release Version Control (SSI-QF-20J) • Copy changelog to Build Numbers Summary (SSI-QF-20L) • Software Checklist (SSI-QF-20K)

Handling Common Scenarios

When Requirements Change

1. Document the change in SSI-QF-10G (Design Change Record)
2. Update SSI-QF-20P (Software Requirements) and SSI-QF-20Q (Functional Specification)
3. Update SSI-QF-20C (Requirements Traceability)
4. Assess impact on existing code and tests
5. Plan implementation in next sprint

When You Discover a Bug

1. Document in SSI-QF-10P (Problem Report)
2. Assess severity and impact
3. For critical bugs:
    - Fix immediately
    - Verify fix doesn't affect other functionality
    - Update documentation
4. For non-critical bugs:
    - Add to product backlog
    - Address in upcoming sprint

When Implementing a Risk Control

1. Identify the risk control in SSI-QF-20C (Requirements Traceability)
2. Implement the control with appropriate testing
3. Document verification evidence
4. Update risk documentation

When Preparing for a Release

1. Ensure all test reports (SSI-QF-20F) are complete
2. Update SSI-QF-20J (Release Version Control)
3. Complete SSI-QF-20K (Software Checklist)
4. Document changes in SSI-QF-20L (Build Numbers Summary)
5. Participate in final design review (SSI-QF-10C)

Best Practices for Medical Device Firmware

Code Quality

- Write clear, maintainable code
- Follow coding standards
- Keep functions small and focused
- Use meaningful variable and function names
- Comment complex algorithms and decisions

Error Handling

- Implement robust error detection
- Use defensive programming techniques
- Never ignore errors
- Log critical errors
- Implement appropriate recovery mechanisms
- Compiler warnings are treated as error

Resource Management

- Optimize for power efficiency
- Manage memory carefully
- Consider real-time constraints
- Handle resource limitations gracefully

Testing

- Test boundary conditions
- Test error cases
- Automate tests where possible
- Document test results thoroughly

Risk Management in Daily Development

Identifying Risks

During development, be constantly aware of potential risks:

- New failure modes introduced by code changes
- Dependencies on external components
- Resource constraints
- Timing issues
- Security vulnerabilities

Documenting Risks

When you identify a potential risk:

1. Discuss with the team
2. Document using appropriate risk management forms
3. Implement and verify risk controls
4. Update traceability matrix

Agile Practices

User Stories and Requirements

- Format: "As a [user], I want [feature] so that [benefit]"
- Link user stories to formal requirements
- Ensure stories have clear acceptance criteria
- Break stories into technical tasks

Definition of Done

A task is only complete when:

- Code is implemented and tested
- Code review is complete
- Documentation is updated
- Traceability is maintained
- Tests pass
- Acceptance criteria are met

Continuous Integration

- Integrate code frequently
- Run automated tests on integration
- Address failures immediately

QMS Documents

QMS Documents for Development

Required QMS Documents and Update Schedule

ID	Name	Contents	When to Update	Responsibility
SSI-QF-20A	Software Safety Classification	• Software safety class (A, B, or C) • FDA "Level of Concern" (Minor, Moderate, Major) • Rationale for classification • Risk analysis	• At project start • When new hazards are identified • When risk controls change	Software Manager
SSI-QF-20B	Software Development Plan	• Development process description • Deliverables for each phase • Verification approach • Risk management approach • Configuration management process	• At project start • When project scope changes • When development approach changes	Project Lead
SSI-QF-20C	Software Requirements Traceability	• User requirements • Software requirements • Mapping to design outputs • Verification methods • Verification results	• When requirements are added/changed • After verification activities • At each sprint review	Firmware Developer
SSI-QF-20D	Software Architecture Design	• System components • Interfaces between components • Hardware interfaces • SOUP components	• After initial design sprints • When architecture changes • Before design reviews	Firmware Developer

SSI-QF-20E	Software Test Protocol	• Test environment • Test procedures • Acceptance criteria • Tests for each requirement	• Before testing begins • When requirements change • When test approach changes	QA Team
SSI-QF-20F	Software Test Report	• Test results summary • Pass/fail status • Anomalies and deviations • Overall assessment	• After test execution • After bug fixes • Before release	QA Team
SSI-QF-20G	Software Maintenance Plan	• Feedback procedures • Problem resolution process • Configuration management • Cybersecurity management	• Before product release • When maintenance procedures change	Software Manager
SSI-QF-20J	Software Release Version Control	• Build numbers • Release dates • Verification status	• At each software release	Software Manager
SSI-QF-20K	Software Checklist	• Pre-release verification checklist • Check for required documentation • Verification of all requirements	• Before software release	QA Team
SSI-QF-20L	Software Build Numbers Summary of Changes	• List of changes in each build • Issues resolved • Known issues	• At each software release	Firmware Developer
SSI-QF-20M	Firmware Programmable Part Release	• Hardware components with firmware • Firmware versions • Verification status	• When firmware is loaded to hardware • For each firmware release	Hardware & Firmware Developers

SSI-QF-20P	Software Requirements Specification	• Functional requirements • Performance requirements • Interfaces • Security requirements	• During initial planning • When requirements change	Software Manager
SSI-QF-20Q	Software Functional Specification	• Detailed functional behavior • Implementation details • Technical specifications	• After requirements are defined • When implementation approach changes	Firmware Developer
SSI-QF-20R	Software User Requirements	• User needs • Use scenarios • User interfaces • User expectations	• During initial planning • When user needs change	Software Manager
SSI-QF-20S	Software Analysis Verification	• Analysis of verification results • Verification of risk controls • Overall software verification	• After verification activities • Before software release	QA Team
SSI-QF-10C	Design Review	• Review participants • Items reviewed • Issues identified • Action items	• At formal design reviews • At key development milestones	Project Lead
SSI-QF-10G	Design Change Record	• Change description • Impact analysis • Verification approach • Approval status	• When design changes are needed • When requirements change	Initiator of Change
SSI-QF-10P	Problem Report	• Problem description • Analysis • Resolution • Verification	• When problems are identified • When problems are resolved	Any Team Member

Document Priority During Development Phases

Initial Planning Phase

- SSI-QF-20A (Safety Classification)
- SSI-QF-20B (Development Plan)
- SSI-QF-20R (User Requirements)
- SSI-QF-20P (Software Requirements)

Design Phase

- SSI-QF-20C (Requirements Traceability)
- SSI-QF-20D (Architecture Design)
- SSI-QF-20Q (Functional Specification)
- SSI-QF-10C (Design Review)

Implementation Phase

- SSI-QF-20E (Test Protocol)
- SSI-QF-10G (Design Change Record) - as needed
- SSI-QF-10P (Problem Report) - as needed

Verification Phase

- SSI-QF-20F (Test Report)
- SSI-QF-20S (Analysis Verification)
- SSI-QF-10C (Design Review)

Release Phase

- SSI-QF-20J (Release Version Control)
- SSI-QF-20K (Software Checklist)
- SSI-QF-20L (Build Numbers Summary)
- SSI-QF-20M (Firmware Part Release) - if applicable

Maintenance Phase

- SSI-QF-20G (Maintenance Plan)
- SSI-QF-10P (Problem Report) - as needed
- SSI-QF-10G (Design Change Record) - as needed

Software Comparison

Development Management Software Comparison: Jira vs YouTrack

Summary

For your firmware development project requiring medical device QMS compliance, **YouTrack** is a more suitable choice due to its simplicity, built-in knowledge base functionality, and cost-effectiveness. One important factor is its robust GitHub integration.

Key Comparison Areas

1. Simplicity of Use

YouTrack ✅ Clear Winner

- YouTrack is designed to be more approachable, for flexibility and simplicity
- Faster onboarding with minimal configuration required
- Modern, clean interface that doesn't overwhelm users
- Less complex setup process for agile workflows

Jira ⚠ More Complex

- Steeper learning curve, many features can be overwhelming
- Requires significant configuration and customization expertise
- Jira offers a comprehensive solution, rich in features and customization options, tailored for larger teams and more complex projects
- Often needs dedicated administrator for optimal setup

2. Wiki Integration & Knowledge Management

YouTrack ✅ Built-in Advantage

- **Integrated Knowledge Base**: YouTrack includes a built-in Knowledge Base that allows you to build a collection of articles, share your team's collective knowledge internally or build a public knowledge base with guides and FAQs

- **Rich Content Support**: Create clear and engaging content with embedded videos and GIFs, add content from external sources like Google Workspace Apps or Figma files, embed Mermaid diagrams, math formulas
- **Project-specific Documentation**: Each article is associated with a specific project, so each team can accumulate and share bits of information that help them accomplish their project goals
- **Version Control**: Knowledge Base stores drafts as you work and keeps a revision history for every article you create
- **Advanced Organization**: Create article trees with unlimited nested hierarchy, browse articles using filters, tags, and advanced query language searches
- **QMS**: Can be used for regulatory documentation

Jira ⚠ Requires Separate Tool

- **No Built-in Wiki**: Jira itself has no native wiki functionality
- **Confluence Dependency**: Requires separate Confluence license and setup for documentation needs → Additional Cost
- **Integration Complexity**: While Jira and Confluence integrate well, it's still two separate systems to manage
- **Learning Curve**: need to learn both Jira and Confluence interfaces

3. GitHub Integration

Both platforms offer GitHub integration, but with different approaches:

YouTrack ✅ Solid Integration

- GitHub integration enables applying commands to YouTrack issues in commit messages, tracking commits in issue activity streams, displaying pull request status, and adding links to issues in commit messages or branch names
- **VCS Integration Setup**: Needs personal access token
- **Smart Commits**: Support for commit message commands that can transition issues and add comments
- **Branch & PR Tracking**: Full visibility of GitHub activity within YouTrack issues
- **Webhook Support**: Real-time synchronization between GitHub and YouTrack

Jira ✅ Feature-Rich Integration

- GitHub for Jira integration allows smart commits, branch and pull request creation directly from Jira, repository indexing with webhooks
- **Comprehensive Tracking:** Connect code changes, pull requests, and branches to Jira issues, track GitHub commits, branches, and pull requests right in Jira
- **Bidirectional Workflow**: Create branches and pull requests on GitHub repositories directly from Jira
- **Advanced Automation**: More sophisticated automation options through smart commits
- **Free Integration**: Atlassian's GitHub for Jira integration is free

Integration Verdict: Both platforms provide excellent GitHub integration. Jira has slightly more advanced features, but YouTrack's integration is more than sufficient for firmware development needs.

Additional Considerations

Cost Comparison

- **YouTrack**: free for up to 10 users
- **Jira**: free for 10 users

Recommendation

Choose YouTrack because:

1. **Simplicity Advantage**: Faster team adoption with less administrative overhead. Easier to train QA
2. **Built-in Knowledge Base**: QMS documentation, SOPs, and regulatory requirements without additional licensing costs
3. **Medical Device Focus**: Single system approach reduces compliance complexity
4. **Adequate GitHub Integration**: Meets all technical requirements for firmware development workflow
5. **JetBrains Ecosystem**: If using JetBrains IDEs, seamless integration enhances developer experience