ENGINEERING FIELD GUIDE

Developing and Manufacturing Medical Devices

A practical guide for embedded software engineers

From software architecture and risk control to verified firmware, secure production release and post-market support

General cross-market guidance | Edition 1.0 | August 2026

Table of Contents

How to use this guide

This guide explains how embedded software engineering fits into the regulated medical-device lifecycle. It is intended for engineers developing firmware, real-time software, device control logic, connectivity, bootloaders, production-programming tools and other software integrated into medical devices or in vitro diagnostic equipment. It applies to both in-house development and outsourced software, components or manufacturing activities.

CORE MESSAGE Working firmware is not yet a compliant medical-device software product. The software must be derived from controlled requirements, contribute to an acceptable device risk profile, be appropriately verified, be reproducibly built and released, be securely provisioned, and remain traceable throughout manufacture, maintenance and change.

Scope and important limitations

- The manufacturer must determine the intended use, intended users, target markets, regulatory classification, applicable legislation, conformity-assessment route and relevant standards for the specific device.
- IEC 62304 concerns medical-device software lifecycle processes; it does not replace system-level risk management, usability engineering, product safety, clinical or performance evidence, cybersecurity, or device-level verification and validation.
- IEC 62304 software safety class, device regulatory classification, FDA software documentation level and cybersecurity exposure describe different matters and must not be treated as interchangeable.
- Standards editions, amendments, national deviations, recognised or harmonised status and regulatory guidance change. Confirm the project's approved standards and regulatory strategy before applying any requirement.
- This guide supports engineering judgement but does not replace the organisation's quality management system, approved development plans, risk management file or applicable legislation and standards.

1. What changes when software becomes part of a medical device

Embedded medical-device software is part of a regulated product, not an isolated coding exercise. Its behaviour can determine treatment delivery, measurement accuracy, alarms, user decisions, stored records and the effectiveness of risk controls. Consequently, source code alone is insufficient: reviewers must be able to reconstruct what the software was intended to do, why its architecture is appropriate, how it was verified, which version was released, and how it behaves when assumptions fail.

The five simultaneous engineering objectives

Objective	What it means for the embedded software engineer
Functional performance	Software implements specified behaviour, timing, accuracy, operating modes and hardware interfaces under realistic operating and fault conditions.
Safety and risk control	Software-related causes, hazardous situations and risk controls are visible, allocated, implemented, independently reviewed and verified.
Secure lifecycle operation	Threats, exposed interfaces, vulnerabilities, credentials, updates, dependencies and production secrets are managed throughout the supported product life.
Reproducible release and manufacture	The approved source, dependencies, toolchain, build configuration, signed image and provisioning process can be identified and reproduced.
Objective evidence	Requirements, architecture, reviews, tests, anomalies, releases and changes are documented, traceable, approved and retrievable.

Design assurance is engineering made reviewable

Useful lifecycle records explain the engineering decisions that matter: required behaviour, hardware assumptions, state transitions, fault handling, timing margins, data integrity, security boundaries, integration behaviour, test outcomes and residual anomalies. Documentation created only after implementation often reveals that requirements, risk controls and verification decisions were not controlled when they should have been.

PRACTICAL TEST A competent engineer outside the original team should be able to identify the released firmware, recreate its build, understand its safety-relevant behaviour, locate the evidence supporting its release and assess the impact of a proposed change.

2. Lifecycle overview and engineering evidence

Development may be iterative, incremental or agile, but medical-device software still requires controlled inputs, defined outputs, appropriate reviews, configuration management and objective evidence. Agile delivery changes the cadence of planning and verification; it does not remove lifecycle obligations.

Lifecycle activity	Typical embedded-software evidence
Planning and classification	Development and verification plans; lifecycle approach; safety-class rationale; security activities; tools; review and approval responsibilities.
Requirements and architecture	Software requirements; interfaces; architecture; software items; segregation; allocation of risk controls and security requirements.
Implementation and unit verification	Controlled source; coding rules; reviews; static analysis; unit tests; defect records; toolchain and dependency baselines.
Integration and system verification	Integration strategy; hardware/software tests; traceability; fault injection; timing, robustness and cybersecurity evidence.
Release and transfer	Approved build; bill of materials; release decision; anomaly assessment; signed images; manufacturing instructions and provisioning controls.
Maintenance and post-market	Change assessments; complaint investigations; vulnerability monitoring; regression evidence; updates and supported-version records.

The evidence chain

1. **Identify the need.** Understand intended use, device function, user interaction and the system or risk-control requirement.
2. **Specify the behaviour.** Write measurable software and interface requirements, including failure responses and acceptance criteria.
3. **Allocate and implement.** Locate the requirement in the architecture, software item, source baseline and configuration.
4. **Verify objectively.** Execute approved review, analysis or test methods against identified versions and record the outcome.

5. **Release and maintain.** Approve the complete configuration, preserve residual-anomaly decisions and reassess later changes or field information.

EVIDENCE PRINCIPLE A test result has limited value if it cannot be connected to an approved requirement, an identifiable software and hardware configuration, a controlled method and a documented acceptance decision.

3. Intended use, regulatory strategy and standards planning

Software decisions depend on what the complete medical device is intended to achieve, who uses it, where it is used, what information or treatment it produces, and what harm could follow incorrect operation. Establish these inputs before fixing architecture, software class, update strategy or verification scope.

Questions to settle early

- What is the intended medical purpose, patient population, intended user, operating environment and expected service life?
- Which device functions rely on software for measurement, therapy, alarms, diagnosis, data transfer, access control or risk reduction?
- What happens if firmware produces an incorrect result, responds late, stops unexpectedly, accepts invalid data or enters the wrong state?
- Which regulations, software lifecycle standards, cybersecurity expectations and product-specific standards apply in each target market?
- What hardware variants, accessories, connectivity options, production environments and update channels must the software support?
- Which parts are developed internally, supplied by third parties, inherited from a platform, generated by tools or reused from another product?

Different classification decisions serve different purposes

Decision	What it describes	Common mistake
Device regulatory class	The applicable product classification and conformity or clearance route in a particular jurisdiction.	Assuming device Class II or IIb determines IEC 62304 software class.

Decision	What it describes	Common mistake
IEC 62304 safety class	The severity of harm to which the software system could contribute, considering appropriately justified risk controls.	Treating Class A, B or C as a general quality score.
FDA documentation level	The scope of software evidence recommended for the relevant US premarket submission.	Equating Basic or Enhanced documentation with IEC 62304 Class A, B or C.
Security exposure	The applicable threats, connectivity, assets, attack surface and security obligations.	Assuming a low software safety class means cybersecurity is unimportant.

For the United States, the FDA Quality Management System Regulation became effective on 2 February 2026 and incorporates ISO 13485:2016 by reference, together with additional FDA requirements. Applicable FDA software and cybersecurity guidance should be selected through the regulatory strategy rather than assumed from the IEC 62304 class.

4. Software safety classification and lifecycle planning

Understand the IEC 62304 safety classes

Class	General interpretation	Engineering implication
A	No injury or damage to health is possible from the hazardous situations to which the software system can contribute.	Document and support the rationale; maintain appropriate requirements, controls, verification and configuration management.
B	Non-serious injury is possible, but serious injury or death is not.	Apply the required additional lifecycle activities, including architecture and appropriate integration and risk-control evidence.
C	Death or serious injury is possible.	Apply the fullest relevant lifecycle rigour, including detailed design where required, unit verification, segregation analysis and stronger evidence.

Classification is based on the potential harm associated with hazardous situations to which the software system can contribute; it is not based simply on the probability that a software defect will occur. Any risk control used to justify a lower classification must be

genuinely independent where necessary, sufficiently effective, documented in the risk management file and verified.

CLASSIFICATION CAUTION An assertion that users will notice an error, that faults are unlikely, or that the software has always worked is not an adequate safety-class rationale. Establish the hazardous sequence, identify credible independent controls and retain the supporting evidence.

Plan the software lifecycle before execution

- **Development planning:** define lifecycle model, activities, deliverables, roles, reviews, safety class, interfaces with system engineering and maintenance expectations.
- **Verification planning:** identify unit, integration and software-system verification; independence; methods; environments; coverage rationale; defect handling and release criteria.
- **Configuration planning:** control source repositories, branches, tags, build scripts, compiler settings, dependencies, test assets, target hardware and released binaries.
- **Risk and security planning:** link safety risk management, threat modelling, secure development, vulnerability handling and security verification to the product lifecycle.
- **Tool and supplier planning:** identify development, build, analysis and test tools, supplier responsibilities, third-party software and the evidence needed to rely on them.

Segregation must be designed and demonstrated

Where software items are assigned different safety classes, show that lower-class items cannot adversely affect higher-class behaviour through memory corruption, processor starvation, shared peripherals, messaging, update mechanisms or common data. MPU or MMU settings, partitioning, watchdog supervision, resource budgets and fault tests may support the segregation argument; a diagram alone does not establish independence.

5. Requirements, architecture and traceability

Good software requirements are testable

A useful software requirement states the triggering condition, required response, timing or accuracy, relevant operating state and acceptance criteria. Avoid wording such as “robust”, “user friendly”, “real time” or “handle errors appropriately” unless those terms are quantified.

Weak requirement	Improved requirement
The device shall detect low battery.	When measured supply voltage remains below the approved threshold for the defined debounce period, the firmware shall issue the specified warning and inhibit the affected therapy function within the specified response time.
The firmware shall recover safely after reset.	After an unexpected reset, the firmware shall keep the actuator disabled, validate retained treatment state and configuration integrity, and require the specified recovery condition before therapy can restart.
Communication shall be secure.	The device shall authenticate the approved peer, use the specified protected transport, reject unauthorised commands and record security-relevant connection failures.

Sources of embedded-software requirements

- Intended use, user needs, system requirements, product claims and expected clinical or diagnostic workflow.
- Hardware interfaces, sensor characteristics, actuator behaviour, calibration, supply conditions and manufacturing configuration.
- Safety risk controls, essential-performance allocations where applicable, alarms, fail-safe behaviour and hazardous-state prevention.
- Cybersecurity risk controls, authentication, authorisation, protected communication, update integrity and logging.
- Environmental exposure, timing, resource capacity, interoperability, usability, service, production test and maintenance needs.
- Applicable regulations, product standards, platform constraints, supplier specifications and identified third-party software limitations.

Architecture should expose critical decisions

- Define software-system boundaries, software items, execution contexts, operating modes, state machines and hardware dependencies.
- Identify safety-related paths, security boundaries, privilege levels, trusted components, shared resources and critical interfaces.
- Allocate requirements and risk controls to software items and identify assumptions placed on hardware, users, accessories or external systems.

- Describe task scheduling, interrupts, memory ownership, synchronisation, communication, error propagation and recovery strategy.
- Record third-party components, boot sequence, secure update path, manufacturing mode, service access and diagnostic information.

Traceability must work in both directions

Maintain usable links from system need or risk control to software requirement, architecture item, implementation or configuration, verification activity, result and release baseline. Reverse traceability should identify requirements and risks affected by a defect, component update, code change or field complaint. The tool may be a managed lifecycle platform, issue tracker or controlled matrix; the important point is reliable, current evidence.

6. Risk management for embedded software

Software contributes to device risk through behaviour and interactions rather than through random physical failure alone. Identify how a defect, invalid assumption, external input, interface failure or compromised component could initiate or fail to prevent a hazardous sequence.

Common software-related hazard contributors

- Incorrect calculations, unit conversions, thresholds, calibration coefficients, rounding or numerical overflow.
- Unexpected task scheduling, race conditions, deadlocks, starvation, priority inversion or missed deadlines.
- Corrupted memory, invalid persistent state, stack overflow, buffer overrun, stale data or uninitialised variables.
- Incorrect mode transitions, actuator commands, alarm suppression, treatment parameters or restart behaviour.
- Sensor disconnection, implausible readings, delayed communications, duplicate messages or loss of time synchronisation.
- Power interruption, battery depletion, brownout, reset, interrupted update or incomplete provisioning.
- Unauthorised access, altered firmware, exposed credentials, insecure debug interfaces or vulnerable third-party components.
- Usability-related misunderstanding, misleading displays, inappropriate defaults, service-mode exposure or ambiguous fault indication.

Follow the risk-control hierarchy

1. **Inherently safe design.** Remove hazardous capabilities where possible, bound outputs, simplify state transitions and prevent invalid configurations.
2. **Protective measures.** Use independent hardware limits, monitoring, plausibility checks, interlocks, watchdogs and verified safe-state behaviour.
3. **Information for safety.** Provide necessary warnings, instructions, service information and limitations when residual risk cannot be addressed more effectively.

Risk-control requirements need measurable evidence

Every software-implemented risk control should be identifiable in the system risk analysis, represented by a specific software requirement, allocated to an architecture element and verified under relevant normal and fault conditions. Include activation thresholds, response time, safe state, reset or recovery conditions, alarm behaviour and interactions with independent protective measures.

Risk analysis is not the same as an FMEA score

FMEA, fault-tree analysis, hazard analysis, state-machine review and fault injection can all contribute useful information, but none replaces the complete device-level ISO 14971 risk-management process. Software defect occurrence probabilities are rarely supported by meaningful statistical evidence; avoid using an optimistic occurrence score to make a severe software-related hazardous situation appear acceptable.

SAFETY/SECURITY INTERFACE A cybersecurity event can initiate a safety hazard, disable a risk control or corrupt clinical information. Link relevant security threats to device safety risks without assuming that every security finding automatically creates patient harm.

7. Detailed embedded-software engineering considerations

7.1 State machines, modes and treatment control

- Define permitted states, entry conditions, exit conditions, transitions, timeouts and the behaviour of every output in each state.
- Prevent invalid state combinations, unintended repeated actions, stale commands and treatment continuation after reset or fault.
- Separate normal, calibration, manufacturing, service, update and emergency modes; control authorisation and transitions between them.

- Verify boundary conditions, interrupted operations, cancellation, retries, loss of communication and return to a known safe state.

7.2 Real-time scheduling, interrupts and resource budgets

- Identify deadlines, task priorities, interrupt rates, worst-case execution time and the effect of shared resources or blocking operations.
- Analyse processor utilisation, stack margins, heap behaviour, queue capacity and timing under peak and degraded operating conditions.
- Control priority inversion, concurrency, re-entrancy, atomic access and lock ordering; prefer bounded, reviewable behaviour.
- Measure timing on representative target hardware rather than relying solely on a simulator or a lightly loaded bench configuration.

7.3 Hardware abstraction, sensors and actuators

- Specify interfaces, units, scaling, sampling, timing, filtering, calibration and tolerance assumptions shared with electronics and systems engineers.
- Handle open circuit, short circuit, saturation, out-of-range values, missing samples, stale data and contradictory sensor inputs.
- Define actuator limits, command validity, feedback checks, interlocks and safe behaviour when the application or communication fails.
- Control differences between development boards, prototype hardware, production revisions and authorised alternative components.

7.4 Memory, numerical behaviour and persistent data

- Control integer overflow, sign conversion, floating-point assumptions, rounding, saturation and unit conversion.
- Define ownership, lifetime and bounds for buffers, queues, stacks and dynamically allocated memory.
- Protect persistent configuration, treatment records and calibration values using appropriate integrity checks, versioning and recovery logic.
- Evaluate flash endurance, partial writes, corruption, factory reset, migration between software versions and behaviour after unexpected power loss.

7.5 Power, reset, watchdogs and recovery

- Define cold start, warm start, brownout, watchdog reset, low-battery shutdown and restoration-of-power behaviour.

- Ensure actuators and safety-critical outputs assume controlled values before the application is fully initialised.
- Use watchdog architecture that detects meaningful application failure; avoid simplistic servicing that can mask a stalled control function.
- Preserve or invalidate retained state deliberately and verify interrupted treatment, interrupted storage and interrupted update scenarios.

7.6 Connectivity, interoperability and external commands

- Define protocol versions, framing, message boundaries, retries, timeouts, duplicate handling and compatibility assumptions.
- Validate incoming data before use and distinguish communication loss from a valid zero, unchanged or unavailable measurement.
- Authenticate authorised peers and constrain which operating modes accept configuration, control, service or update commands.
- Verify degraded networks, pairing changes, simultaneous connections, malformed messages and interactions with apps, cloud services or accessories.

7.7 Secure design, credentials and update mechanisms

- Identify assets, trust boundaries, entry points, threat scenarios and security objectives early enough to influence the architecture.
- Protect sensitive data at rest and in transit using appropriate controls; protect data in use through least privilege, process or memory isolation, trusted execution or other justified measures where relevant.
- Use authenticated boot and authorised, integrity-protected firmware updates where justified by the threat model and intended operating environment.
- Protect production keys and device credentials, prevent insecure defaults, control debug access and assess rollback or recovery behaviour.
- Monitor third-party vulnerabilities, maintain an appropriate SBOM and establish vulnerability disclosure, triage, remediation and communication processes.

IMPORTANT DISTINCTION “Encryption in use” is not a universal practical requirement for embedded firmware: processors often need plaintext to perform their function. Define the actual confidentiality threat and justify appropriate controls such as secure enclaves, protected memory, isolated execution or restricted access.

7.8 Coding standards, reviews and static analysis

- Adopt a justified coding standard for the chosen language and platform; define deviation handling and review expectations.
- Review safety-critical logic, error handling, concurrency, arithmetic, interfaces, security controls and assumptions rather than only code appearance.
- Use compiler warnings, static analysis, dependency scanning and other appropriate automated checks with controlled versions and dispositions.
- Record review outcomes, findings, accepted deviations and remediation in a form linked to the relevant baseline or change.

7.9 Diagnostic logging and field investigation

- Identify which resets, faults, state transitions, configuration changes and security events need to be recorded for service or investigation.
- Avoid exposing patient data, credentials, secrets or security-sensitive implementation details in diagnostic logs.
- Define timestamps, retention, storage limits, integrity, retrieval permissions and behaviour when logging resources become full.
- Verify that diagnostic functionality cannot degrade treatment performance, compromise safety controls or create an unauthorised service pathway.

8. Design outputs, third-party software and configuration control

Minimum controlled embedded-software package

- Approved software development and verification plans, software safety-class rationale and applicable lifecycle procedures.
- Software requirements, interface definitions, architecture, relevant detailed design and risk-control allocation.
- Controlled source code, configuration files, generated assets, build scripts, compiler and linker settings, and toolchain identification.
- Third-party component inventory, SOUP assessment, licences, supplier evidence and SBOM appropriate to the device and target market.
- Review records, unit and integration tests, static-analysis output, software-system verification, anomalies and traceability.
- Release package containing the approved binary, version, hash, signature where applicable, compatibility information and residual-anomaly assessment.

- Manufacturing-programming, provisioning, production-test, installation, service and update instructions as applicable.

Control SOUP and other third-party dependencies

Software of unknown provenance may include an RTOS, bootloader, vendor SDK, communications stack, cryptographic library, middleware or prebuilt binary for which full development evidence is unavailable. Identify the component and exact version, intended use, required functionality, known anomalies, supplier information, security exposure and the verification needed to support reliance on it.

Dependency type	Typical engineering evidence
RTOS or scheduler	Version, configuration, task behaviour, known defects, timing assumptions, resource usage and relevant integration tests.
Vendor SDK or hardware driver	Supported silicon and revisions, errata, peripheral assumptions, generated-code settings and fault-handling tests.
Wireless or protocol stack	Version, interoperability, exposed services, security configuration, malformed-input behaviour and update support.
Cryptographic component	Approved algorithms, configuration, entropy assumptions, key handling, known vulnerabilities and supported lifecycle.
Build or test tool	Tool identity, intended use, impact on evidence, confidence measures and configuration or validation as appropriate.

A configuration baseline includes more than source code

A release baseline should identify the repository commit, controlled branch or tag, dependency versions, submodules, generated files, compiler and linker versions, optimisation settings, build scripts, hardware revision, configuration data, cryptographic signing identity, binary hash and associated test evidence. A rebuild that silently changes any of these inputs is not necessarily the same device software configuration.

REPRODUCIBILITY CHECK If the original build computer disappeared, could an authorised engineer recreate the released image, identify any legitimate non-determinism and demonstrate that the tested configuration matches the manufactured product?

9. Reviews, prototypes and design maturity

Prototype software has a defined purpose

Stage	Typical purpose	Evidence limitation
Exploratory proof of concept	Investigate hardware behaviour, timing feasibility, algorithms or user interaction.	Useful for learning, but not automatically controlled or representative verification evidence.
Integrated engineering build	Develop architecture, interfaces, risk controls, diagnostics and representative hardware interaction.	Configuration, known deviations and hardware maturity must be recorded before results are reused.
Verification candidate	Provide a controlled baseline suitable for approved software and system verification.	Open anomalies, build changes and non-representative dependencies need formal assessment.
Release candidate	Support final release, manufacturing transfer, compatibility confirmation and residual-risk review.	A release decision requires complete traceability and controlled disposition of remaining issues.

Technical reviews should address real failure modes

- Review requirements for completeness, measurability, fault responses, timing and traceability to system needs and risk controls.
- Review architecture for segregation, state transitions, scheduling, shared resources, security boundaries and hardware assumptions.
- Review source changes for safety impact, concurrency, arithmetic, error handling, defensive behaviour and coding-standard deviations.
- Review verification readiness for representative builds, target hardware, tools, test data, environment and objective acceptance criteria.
- Review release readiness for anomalies, cybersecurity, manufacturing compatibility, documentation and approved configuration.

Independence should match the activity and risk

The degree of review and verification independence should follow applicable procedures, safety class, regulatory expectations and the risk significance of the item. Independence may involve another competent engineer, a separate verification function or

organisational separation; merely changing the name on a form does not create an independent technical challenge.

10. Software verification, integration and validation support

Verification and validation answer different questions

Activity	Question answered	Typical embedded-software role
Unit verification	Does an individual software unit or small component satisfy its design and specified behaviour?	Test logic, boundaries, calculations, state transitions and abnormal inputs using appropriate target or host methods.
Integration verification	Do combined software items and hardware interfaces operate correctly together?	Exercise scheduler interactions, drivers, communications, shared data, fault propagation and timing.
Software-system verification	Does the integrated software system fulfil its approved software requirements?	Verify functional behaviour, risk controls, operating modes, robustness and software-specific acceptance criteria.
Device validation	Does the finished medical device meet user needs and intended use?	Provide representative released software and support realistic workflow, usability, clinical or performance evaluations.

Build a defensible verification protocol

- Identify the approved requirement, risk control or architectural behaviour being verified.
- Define the software version, binary hash, hardware revision, configuration, third-party components and relevant operating environment.
- State the method, test setup, input data, preconditions, equipment, tools and objective acceptance criteria.
- Include normal operation, boundary conditions, representative use, invalid input and credible fault conditions.
- Record the result, observed behaviour, reviewer or tester, date, anomalies, deviations and final disposition.
- Maintain traceability from failed and repeated tests to the defect, corrective change, new configuration and regression decision.

Select methods for the specific failure mechanism

- Use requirements-based tests, interface tests, state-machine coverage, boundary-value analysis and equivalence partitioning where appropriate.
- Apply fault injection to lost sensors, corrupted messages, power interruption, memory errors, watchdog events and invalid persistent data.
- Measure timing, resource margins and long-duration behaviour on representative target hardware.
- Use static analysis, source review, compiler diagnostics, dependency assessment and security-focused testing to complement execution tests.
- Consider fuzzing, penetration testing, update tampering, authentication bypass attempts and malformed external input according to the threat model.

Coverage needs an engineering rationale

Code coverage can reveal unexercised logic, but a numerical percentage does not demonstrate correct requirements, effective risk controls or meaningful assertions. Select and justify coverage measures according to safety class, architecture, risk, complexity and applicable organisational requirements. Do not state that IEC 62304 universally mandates a particular statement, branch or MC/DC percentage when the project has not established such a requirement.

Manage anomalies without hiding them

Record unexpected behaviour, classify its potential safety, security and performance impact, determine the affected software and hardware configurations, and assess whether regression testing or risk-file updates are required. Repeating a test until it passes does not resolve an intermittent software failure. Any released known anomaly requires a documented rationale, residual-risk evaluation and formal approval.

11. Cybersecurity and secure software lifecycle

Cybersecurity is a lifecycle concern whenever a device, accessory, update path or support environment exposes assets or interfaces that could be misused. The applicable expectations depend on intended use, connectivity, market, architecture and the potential impact on safety and essential device functions.

Establish and maintain the threat model

- Identify sensitive assets, safety-critical functions, patient or diagnostic information, service credentials and signing keys.

- Map trust boundaries, physical and wireless interfaces, manufacturing access, mobile applications, cloud connections and update channels.
- Describe credible threat scenarios, prerequisites, exploited weaknesses, affected assets and potential device or patient impact.
- Allocate security requirements and connect security events to safety hazards where a realistic causal link exists.
- Update the model when interfaces, suppliers, dependencies, deployment assumptions or known vulnerabilities change.

Security controls must be implemented and verified

Control area	Representative embedded-software evidence
Identity and access	Unique device identity; authorised users or services; controlled privilege; debug-lock and manufacturing-mode checks.
Data protection	Protected storage and communication; justified handling of data in use; key lifecycle and sensitive-log controls.
Platform integrity	Boot-chain assumptions, signed or otherwise authenticated images, integrity verification and recovery behaviour.
Update capability	Authorised updates, compatibility checking, interruption handling, rollback decisions and supported-version management.
Dependency management	SOUP assessment, appropriate SBOM, vulnerability monitoring, exploitability assessment and remediation records.
Security monitoring	Security-relevant logging, disclosure intake, vulnerability triage, incident response and customer communication where needed.

Production security is part of the product security model

A strong device architecture can be undermined by shared manufacturing credentials, unprotected signing keys, open programming interfaces, undocumented production overrides or uncontrolled service software. Define who can build, sign, provision, unlock and rework a device, and verify that production-only capabilities do not remain available to unauthorised users after release.

US SUBMISSION REMINDER FDA device-software submission recommendations and FDA cybersecurity recommendations are separate, complementary evidence streams. A complete software requirements and test package does not automatically satisfy cybersecurity design, vulnerability or submission expectations.

12. Design transfer, manufacturing and provisioning

Transfer the software product, not just a binary

Design transfer converts controlled software development outputs into a repeatable production and release process. Manufacturing needs the approved firmware image, version-identification method, compatible hardware configuration, programming instructions, provisioning data, production-test criteria, security controls and failure-handling rules.

Manufacturing readiness checklist

- Approved software release, identified image and hash, and authorised signing or integrity-protection mechanism.
- Documented compatibility with PCB revision, microcontroller variant, bootloader, accessories and production-test equipment.
- Controlled programming tool, firmware loader, scripts, fixtures, operator permissions and workstation configuration.
- Defined serialisation, device identity, calibration, certificates, keys and other per-device provisioning inputs.
- Automated or otherwise controlled confirmation that the intended image and configuration were installed successfully.
- Production-test coverage for critical interfaces, version reporting, configuration integrity, safety-related checks and security state.
- Instructions for interrupted programming, failed provisioning, quarantine, rework, key revocation and disposal of sensitive material.
- Device-history records linking the manufactured unit to released software, relevant hardware, provisioning data and test results.

Control programming and device provisioning

Manufacturing activity	Control objective
Image selection	Prevent use of draft, unapproved, incompatible or superseded firmware.

Manufacturing activity	Control objective
Image programming	Verify write completion, correct image identity and integrity before the device advances.
Device identity	Assign authorised serial number, identifiers, certificates or credentials without duplication or exposure.
Calibration and configuration	Apply approved device-specific data, validate limits and preserve traceability to equipment or reference values.
Security finalisation	Disable unapproved debug or manufacturing access and verify the intended operational security state.
Functional release test	Confirm required startup, communication, interfaces, configuration and safety-relevant production checks.

Manufacturing software and test tools can affect quality

Programming utilities, fixture firmware, calibration applications, automated test scripts and manufacturing execution interfaces can determine whether a nonconforming device is accepted. Identify their intended use and potential impact, control versions and access, assess appropriate validation or confidence measures, and retain evidence that changes do not compromise production acceptance.

Pilot builds should challenge the complete process

Use representative pilot builds to confirm image selection, programming time, hardware compatibility, serialisation, provisioning, secure-key handling, production test, failure recovery, record generation and operator instructions. Investigate unexpected retests, undocumented manual steps, configuration mismatches and any route by which an unapproved image can enter production.

13. Release, nonconformity and production feedback

A controlled release has explicit acceptance criteria

- Required software activities and planned reviews have been completed or any approved deviations have been formally justified.
- Requirements, risk controls and relevant security controls have satisfactory, traceable verification evidence.

- The exact source, toolchain, dependencies, configuration, hardware compatibility and released binary are identified.
- Open anomalies and vulnerabilities have been assessed for safety, performance, cybersecurity and market impact.
- Manufacturing, installation, update, service and customer-facing information is aligned with the released configuration.
- Authorised roles have approved the release, and the resulting package is protected from unauthorised alteration.

Contain first, then investigate

1. **Identify and contain.** Stop affected programming, segregate potentially impacted devices and determine which versions, lots or serial numbers are involved.
2. **Preserve the evidence.** Record firmware identity, logs, hardware revision, configuration, reproduction steps and the original production or field observation.
3. **Assess the impact.** Evaluate intended use, safety risk, cybersecurity, functional performance, released stock and fielded devices.
4. **Define and verify the correction.** Control the software or production change, review it, execute justified regression testing and verify any reprogramming activity.
5. **Decide escalation.** Determine whether corrective action, complaint handling, supplier action, field communication or regulatory reporting is necessary.

Reprogramming and rework are controlled activities

Reflashing a device changes its controlled configuration and may also alter calibration, credentials, records, bootloader state or compatibility. Define authorised images, operator permissions, supported hardware, data-preservation rules, acceptance tests and traceability. Do not treat firmware rework as an informal bench activity simply because it leaves no visible physical change.

Production feedback is software evidence

Trend programming failures, reset rates, boot-time anomalies, calibration rejects, communication failures, fixture overrides, retests and version mismatches. These signals may expose latent timing, hardware-compatibility, configuration or provisioning weaknesses before a complaint reaches the field.

14. Change control, maintenance and post-market support

Every software change needs an impact assessment

- Identify affected requirements, architecture items, risk controls, cybersecurity threats, third-party components and interfaces.
- Assess supported hardware variants, existing production stock, fielded devices, mobile applications, cloud services and accessories.
- Determine the necessary code review, unit testing, integration testing, software-system verification and regression scope.
- Consider software safety classification, open anomalies, performance claims, labelling, update instructions and regulatory submission impact.
- Update configuration records, SBOM, release information, vulnerability assessments, manufacturing tools and service documentation where relevant.
- Confirm that production, service and field update routes can apply the change safely and retain evidence of the resulting configuration.

Post-market embedded-software responsibilities

- Support complaint and service investigations using controlled logs, reproducible builds, configuration records and representative test environments.
- Preserve field evidence before a device is reset, updated, reconfigured or reprogrammed.
- Monitor reported defects, component advisories, security vulnerabilities, unexpected resets, treatment interruptions and communication problems.
- Assess whether new information changes safety risks, the threat model, residual-anomaly decisions or the need for corrective action.
- Maintain supported-version records and clearly identify which devices received updates, configuration changes or field corrections.
- Coordinate safety, quality, regulatory, cybersecurity, operations, manufacturing and supplier responses when software issues cross organisational boundaries.

An unreproduced fault is not a completed investigation

Intermittent embedded-software failures may depend on timing, specific hardware revisions, power transients, state history, corrupted persistent data, network conditions or interactions with other tasks. Investigate available logs, reset causes, firmware and hardware baselines, environmental conditions, accessory combinations and production history before accepting “could not reproduce” as an outcome.

MAINTENANCE MINDSET A released device is a maintained configuration, not a frozen snapshot. Dependency vulnerabilities, supplier changes, field complaints and new attack techniques can change the acceptability of software that originally passed every planned test.

15. Common failure patterns and practical checklists

Common project failure patterns

Failure pattern	Why it causes trouble	Better practice
Coding starts before requirements stabilise	Architecture decisions become hidden requirements and critical fault behaviour is discovered late.	Maintain controlled assumptions, mature requirements by gate and review the architecture against risk early.
Software class is selected by preference	Lifecycle activities and safety evidence may be insufficient for the actual potential harm.	Create a traceable, device-level classification rationale supported by verified independent controls.
Risk controls exist only in the risk file	The required behaviour is missing from software requirements, implementation or verification.	Allocate each control to measurable software requirements and verify the complete hazardous scenario.
Passing unit tests substitute for integration	Timing, hardware interfaces, concurrency and recovery failures remain undetected.	Combine unit, target, integration, fault-injection and software-system verification.
Released images cannot be rebuilt	The tested configuration cannot be connected confidently to the manufactured or fielded product.	Baseline source, dependencies, toolchain, scripts, settings, hardware and signed binary.
Manufacturing credentials are shared	Compromise can expose signing keys, device identity or unrestricted service access.	Apply individual access, protected keys, secure provisioning and controlled production finalisation.
A vulnerability is dismissed without context	An exploitable dependency may remain in safety-relevant or externally reachable firmware.	Assess affected versions, attack path, device impact, compensating controls and remediation.
Intermittent failures are closed as no fault found	Timing, power or state-dependent defects persist in production or the field.	Preserve evidence, investigate system interactions and define responsible closure criteria.

Embedded software engineer's phase checklist

At this point	Confirm before proceeding
Before architecture	Intended use, software contribution, safety class, markets, operating environment, threats, interfaces and preliminary hazards are understood.
Before implementation	Requirements and interfaces are controlled; architecture, segregation, state handling, coding rules and allocated risk controls are reviewed.
Before integration	Units, dependencies, hardware assumptions, build configuration, defect status and relevant unit-verification evidence are identified.
Before formal verification	The baseline is frozen; hardware is representative; protocols, tools, test data, acceptance criteria and risk-control coverage are approved.
Before release and transfer	Traceability is complete; anomalies and vulnerabilities are assessed; approved binaries, signatures, manufacturing tools and instructions are ready.
Before change approval	Requirements, safety, security, hardware compatibility, production, fielded devices, regression scope and regulatory impact are assessed.

ENGINEER'S STOP SIGNAL Pause and escalate when a requested shortcut would obscure the released software configuration, bypass a risk control, conceal an anomaly, expose a production secret, disable security controls or release a device without the required objective evidence.

Closing perspective

The embedded software engineer's contribution extends far beyond writing firmware. It includes translating intended use and system needs into measurable behaviour; designing state, timing, interfaces and fault responses; supporting effective risk and security controls; generating credible verification evidence; transferring a reproducible, secure product into manufacture; and maintaining that product responsibly throughout its commercial life. When these activities are integrated from the start, regulatory evidence becomes the natural output of disciplined engineering rather than a retrospective reconstruction.

Appendix A. Embedded-software deliverables by lifecycle phase

Phase	Core embedded-software deliverables
Planning	Software development and verification plans; safety-class rationale; standards applicability; configuration, security and supplier approaches.
Inputs	Software requirements; hardware/software interfaces; operating modes; timing, safety, security, manufacturing and maintenance requirements.
Architecture	Software items; component boundaries; state machines; task model; segregation; resource assumptions; trust boundaries and allocated risk controls.
Implementation	Controlled source and configuration; detailed design where applicable; coding-rule deviations; reviews; dependency and toolchain records.
Verification	Unit, integration and software-system protocols and reports; test assets; fault injection; security evidence; traceability and anomaly records.
Release	Approved binary, hash or signature, release notes, compatibility matrix, residual-anomaly decisions, SBOM and formal release approval.
Transfer	Programming and provisioning instructions; manufacturing tools; device identity and calibration process; production tests and pilot evidence.
Production	Installed software and hardware identification; serialisation; provisioning records; production-test outcomes; deviations and rework evidence.
Maintenance	Changes, regression results, complaint investigations, vulnerability assessment, updates, supported versions and field-action records.

Appendix B. Commonly relevant standards and frameworks

This list is a starting point, not an applicability determination. Use the project's controlled standards process to confirm the applicable edition, amendment, national adoption, recognised or harmonised status, product-specific requirements and target-market expectations.

Topic	Common reference	Why embedded software engineers care
Quality management	ISO 13485:2016; US FDA 21 CFR Part 820 QMSR	Development controls, review, suppliers, production, records, change control, nonconformity and CAPA.
Software lifecycle	IEC 62304:2006 + A1:2015	Software planning, safety classification, development, verification, maintenance, configuration and problem resolution.
Device risk management	ISO 14971:2019; ISO/TR 24971:2020	Hazards, hazardous situations, risk controls, residual risk and production or post-production feedback.
Secure product lifecycle	IEC 81001-5-1:2021 and applicable interpretations	Security activities integrated with the health-software product lifecycle.
Usability	IEC 62366-1 and relevant product standards	Use-related risk, user interfaces, alarms, workflows and software interactions.
Medical electrical equipment	IEC 60601-1 family where applicable	Basic safety, essential performance and software-relevant requirements for applicable medical electrical equipment.
Laboratory and IVD equipment	IEC 61010 family where applicable	Safety requirements and software-relevant equipment behaviour for applicable laboratory or IVD systems.
US software submissions	FDA Content of Premarket Submissions for Device Software Functions	Software description, risk information, requirements, architecture, verification, anomalies and documentation level.
US cybersecurity submissions	FDA Cybersecurity in Medical Devices guidance and applicable statutory obligations	Threat modelling, secure design, SBOM, vulnerability management, labelling and cybersecurity evidence.
EU medical devices and IVDs	Regulation (EU) 2017/745; Regulation (EU) 2017/746	Applicable general safety and performance requirements, lifecycle evidence, software and cybersecurity considerations.

Appendix C. Glossary

Term	Meaning in this guide
Bootloader	Software that starts the device and may validate, select, install or recover an application image.
CAPA	Corrective and preventive action used to investigate and address actual or potential systemic quality problems.
Configuration baseline	The approved combination of source, dependencies, tools, settings, hardware assumptions and released software artefacts.
Essential performance	Performance necessary to achieve freedom from unacceptable risk where this concept applies under the relevant product standard.
Firmware	Software embedded in a device or hardware component and supplied as part of its controlled product configuration.
IEC 62304 safety class	Class A, B or C assigned according to the potential severity of harm to which the software system can contribute.
Integration verification	Evidence that combined software items and hardware/software interfaces behave correctly together.
Objective evidence	Verifiable records, facts or data demonstrating that a requirement, activity or process has been fulfilled.
Provisioning	Controlled installation of software, identity, configuration, calibration values, certificates or credentials into a device.
Risk control	Measure used to reduce risk; software-implemented controls must be specified, implemented and verified.
SBOM	Software bill of materials identifying relevant software components and dependencies.
SOUP	Software of unknown provenance for which the manufacturer does not have complete medical-device lifecycle evidence.
Software item	An identifiable part of a software system within the defined software architecture.
Software unit	A software item that is not subdivided into other items for the purposes of the chosen design and verification approach.
Threat model	Structured description of assets, interfaces, trust boundaries, threats, weaknesses and security controls.
Traceability	The ability to connect needs, risks, requirements, design, implementation, verification, release and change information.

Term	Meaning in this guide
Validation	Confirmation that requirements for a specific intended use or application have been fulfilled.
Verification	Confirmation that specified requirements have been fulfilled.
VEX	Vulnerability exploitability exchange information describing whether a known vulnerability affects a particular product or component context.

Appendix D. Authoritative starting sources

- ISO 13485:2016 — Medical devices — Quality management systems
- ISO 14971:2019 — Application of risk management to medical devices
- ISO/TR 24971:2020 — Guidance on the application of ISO 14971
- IEC 62304:2006 + A1:2015 — Medical device software lifecycle processes
- IEC 81001-5-1:2021 — Security activities in the product lifecycle
- FDA Quality Management System Regulation (QMSR)
- FDA — Content of Premarket Submissions for Device Software Functions
- FDA — Cybersecurity in Medical Devices: Quality Management System Considerations and Content of Premarket Submissions
- FDA — Off-the-Shelf Software Use in Medical Devices
- Regulation (EU) 2017/745 on medical devices
- Regulation (EU) 2017/746 on in vitro diagnostic medical devices

CONTROLLED USE Consult the controlled, authorised editions of relevant standards and regulatory texts. Public summaries, supplier claims and inherited project templates are useful starting points, not substitutes for the applicable requirements.