WHITEPAPER

Engineering Long-Life Battery-Powered Smart Water Meters

Part 4 — Firmware as the Energy Manager

A Systems Engineering Approach to Battery Life in Advanced Metering Infrastructure

Author	Vincent Wong (Wong Yew Hon)
Published	8th September 2026
Category	Smart Water Metering / AMI / Battery Systems Engineering
Part	4 of 7 — **Firmware as the Energy Manager** Whitepaper Series: ☐ Part 1 Understanding Battery Life and the Battery Life Budget ☐ Part 2 Battery Chemistry and Energy Storage ☐ Part 3 Ultra-Low-Power Hardware Architecture ☑ **Part 4** **Firmware as the Energy Manager** ☐ Part 5 Battery Life Qualification and Validation ☐ Part 6 From Laboratory to Field: Achieving Real-World Battery Life ☐ Part 7 Practical Recommendations and the Future of Long-Life Smart Water Metering
Version	1.0 — First Edition

"Hardware determines how efficiently energy can be consumed. Firmware determines when that energy is consumed."

4.1 Firmware Is the Battery Life Manager

Hardware establishes the electrical foundation of an ultra-low-power smart water meter, but firmware determines how effectively that hardware is used throughout the product's operational life.

Every decision made by the firmware influences battery consumption.

Examples include:

- How frequently the meter wakes from sleep
- How often sensors are sampled
- When data is transmitted
- How communication retries are handled
- When alarms are generated
- How battery measurements are performed
- How firmware updates are managed

Collectively, these decisions define how the Battery Life Budget is allocated over the intended service life.

Even the most energy-efficient hardware cannot achieve long battery life if firmware repeatedly performs unnecessary operations.

Firmware Controls Every Wake-Up

For much of its operational life, a smart water meter remains asleep.

During sleep:

- The processor consumes minimal current.
- Communication modules remain powered down.
- Sensors remain inactive.
- Memory activity is minimal.

Battery energy is consumed very slowly.

Every wake-up event temporarily interrupts this low-power state.

Firmware therefore determines:

- When wake-up occurs
- Why wake-up occurs
- How long the system remains active
- Which peripherals become powered
- When the system returns to sleep

Reducing unnecessary wake-up events is one of the most effective methods of extending battery life.

Every Wake-Up Has an Energy Cost

A wake-up involves much more than processor execution.

A typical wake-up sequence may include:

- Clock stabilisation
- Peripheral initialisation
- Sensor activation
- Battery voltage measurement
- Meter reading
- Memory access
- Communication preparation
- Data transmission
- Logging
- Return to sleep

Each activity consumes energy.

Consequently, firmware designers should regard every wake-up as an engineering decision rather than an automatic process.

Minimise Active Time

Battery life depends not only on how often the processor wakes but also on how quickly it returns to sleep.

Good firmware design aims to:

- Complete tasks efficiently.
- Avoid unnecessary delays.
- Eliminate idle waiting.
- Use interrupt-driven processing where appropriate.
- Shut down peripherals immediately after use.

The objective is not to keep the processor busy.

The objective is to complete useful work using the smallest practical amount of energy.

Event-Driven Rather Than Time-Driven

Traditional embedded systems often rely on fixed polling intervals.

For example:

- Wake every minute.
- Read every sensor.
- Check every input.
- Return to sleep.

While simple to implement, polling may consume energy even when nothing has changed.

Modern smart water meters increasingly adopt event-driven firmware.

Examples include:

- Leak alarm
- Tamper detection
- Reverse flow
- Magnetic interference
- Scheduled reporting
- Remote command

The processor wakes only when useful work needs to be performed.

This significantly improves overall energy efficiency.

Firmware Should Respect the Battery Life Budget

Throughout this whitepaper, we have introduced the concept of the Battery Life Budget.

Firmware is responsible for enforcing that budget.

Whenever new features are proposed, engineers should ask:

- How many additional wake-ups are required?
- How much additional processing is needed?
- Will communication frequency increase?
- Does this feature justify the additional battery consumption?

Treating battery energy as a managed engineering resource encourages disciplined firmware development throughout the product lifecycle.

Engineering Insight

Every firmware feature has an energy cost. Long-life products are achieved not by adding more functionality, but by ensuring every function justifies the battery energy it consumes.

Practitioner Observation

As AMI products mature, firmware naturally evolves through bug fixes, cybersecurity enhancements, new communication features, and utility-specific requirements. Each software revision has the potential to alter energy consumption. Battery life should therefore be revalidated whenever significant firmware changes are introduced rather than assuming the original battery life estimate remains valid throughout the product's lifecycle.

Key Takeaways

- Firmware determines how the Battery Life Budget is allocated.
- Every wake-up consumes energy.
- Reducing wake-up frequency often delivers larger savings than optimising processor speed.
- Event-driven firmware generally provides greater energy efficiency than continuous polling.
- Battery life should remain a design consideration throughout firmware development.

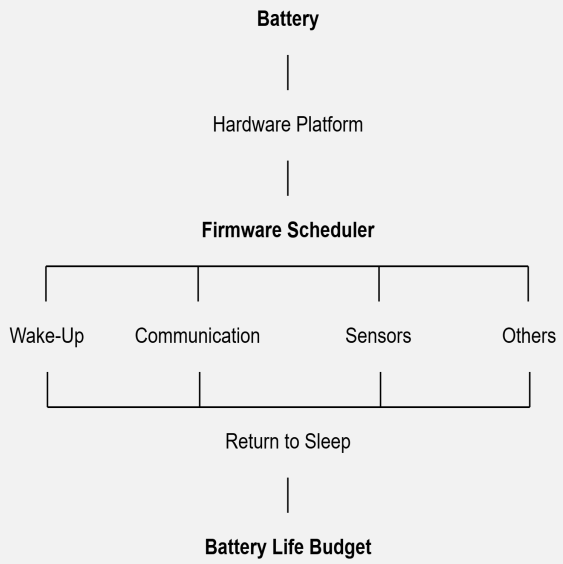

Figure 4.1 — Firmware Controls the Battery Life Budget
Hardware provides the capability for low-power operation, while firmware determines when energy is consumed and how the Battery Life Budget is allocated throughout the product lifecycle.

4.2 Sleep Modes and Wake-Up Strategy

For most of its operational lifetime, a battery-powered smart water meter performs no useful work.

It simply waits.

This observation may appear surprising, but it forms the basis of ultra-low-power product design.

A smart water meter designed for a ten to fifteen-year service life typically spends well over **99% of its time in a low-power sleep state**, waking only briefly to perform scheduled tasks or respond to specific events.

Consequently, one of the most effective methods of extending battery life is not to make the processor run faster—it is to **keep the processor asleep for as long as practical.**

Sleep Is the Normal Operating State

In traditional embedded systems, active operation is often considered the default state.

Long-life battery-powered products reverse this philosophy.

For smart water meters:

- Sleep is the normal operating state.
- Active processing is the exception.

This simple change in design philosophy has profound implications.

Instead of asking:

> "When should the processor sleep?"

Engineers should ask:

> **"What justifies waking the processor?"**

Every wake-up should have a clear operational purpose.

Modern Microcontrollers Provide Multiple Sleep Modes

Ultra-low-power microcontrollers typically offer several power-saving modes.

Examples include:

- Idle mode
- Standby mode
- Stop mode
- Deep sleep
- Backup mode
- Shutdown mode

Each mode offers different trade-offs between:

- Power consumption
- Wake-up latency
- Memory retention
- Peripheral availability
- Clock availability

Selecting the appropriate sleep mode depends on the operational requirements of the product.

Deeper sleep generally reduces energy consumption but may require longer recovery time or additional firmware initialisation after wake-up.

Every Wake-Up Consumes More Than CPU Energy

A common misconception is that wake-up energy is determined solely by processor execution time.

In reality, waking the system often activates multiple hardware subsystems.

A typical wake-up may involve:

- Restoring system clocks
- Starting oscillators
- Enabling voltage regulators
- Powering sensors
- Reading the meter
- Accessing non-volatile memory
- Initialising communication modules
- Performing security checks

Each activity contributes to the total energy consumed during the wake-up cycle.

Therefore, reducing the number of wake-up events often produces greater battery savings than reducing the execution time of individual tasks.

Batch Tasks Whenever Practical

Rather than waking the processor repeatedly for individual activities, firmware should combine multiple operations into a single wake-up session whenever operationally acceptable.

For example, during one wake-up event the firmware may:

- Read the meter.
- Measure battery voltage.
- Process alarms.
- Update internal logs.
- Prepare the communication payload.
- Transmit scheduled data.
- Return immediately to sleep.

This approach reduces the overhead associated with repeated wake-up and shutdown sequences.

Wake Only for Meaningful Events

An effective wake-up strategy distinguishes between events that require immediate action and those that can be deferred.

Examples of **high-priority wake-up events** include:

- Scheduled billing reads

- Leak detection
- Reverse flow alarms
- Tamper detection
- Remote commands
- Firmware updates

Lower-priority maintenance tasks may instead be combined with the next scheduled wake-up to minimise unnecessary energy consumption.

This event-driven approach allows firmware to preserve battery energy without compromising operational requirements.

Avoid "Convenience Wake-Ups"

During product development, engineers may add periodic wake-up events for convenience.

Examples include:

- Frequent status logging
- Diagnostic counters
- Development debug messages
- Repeated battery checks
- Routine self-tests

While individually insignificant, these additional wake-ups may accumulate over many years into a measurable reduction in battery life.

Before adding any recurring activity, engineers should ask:

> **Does this wake-up provide sufficient operational value to justify its energy cost?**

Engineering Insight

> **The processor should remain asleep until there is a meaningful reason to wake it. Every unnecessary wake-up permanently reduces the Battery Life Budget.**

Practitioner Observation

In many AMI projects, firmware gradually evolves as new features are added over several years. Additional diagnostics, security enhancements, telemetry fields, or maintenance functions often introduce extra wake-up events without a corresponding review of their battery impact. Periodic firmware energy audits help ensure that accumulated feature growth does not gradually erode the original battery life target.

Key Takeaways

- Sleep should be regarded as the normal operating state.
- Wake-ups should occur only when justified by operational requirements.
- Wake-up energy includes the entire system, not only the processor.
- Combining multiple tasks into one wake-up reduces overhead.
- Periodic review of firmware behaviour helps preserve long-term battery performance.

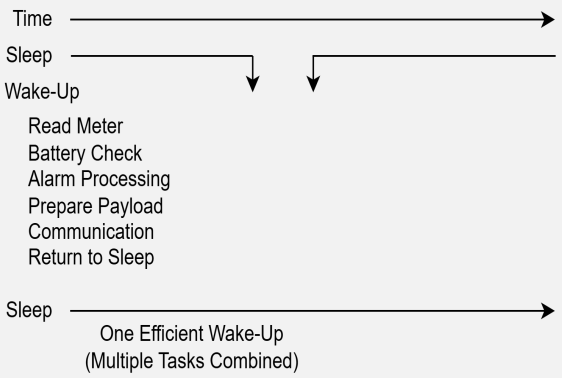

Figure 4.2 — Efficient Wake-Up Strategy
Combining multiple activities into a single wake-up session minimises repeated wake-up overhead and improves overall energy efficiency.

4.3 Scheduling Meter Reading and Communication

For battery-powered smart water meters, **when** data is collected and **when** it is transmitted are just as important as how efficiently the hardware operates.

Meter reading and wireless communication are among the most energy-intensive activities performed by the device.

Consequently, firmware scheduling has a direct influence on battery life.

Poor scheduling can significantly increase energy consumption even when the hardware has been carefully optimised.

Conversely, intelligent scheduling allows the same hardware platform to operate reliably for many years while meeting utility operational requirements.

Meter Reading and Data Transmission Are Different Activities

One of the most common misconceptions is that every meter reading must immediately be transmitted to the utility.

In reality, these are two separate operations.

The firmware may:

- Measure consumption.
- Store the reading locally.
- Process alarms.
- Update internal statistics.

Only later does it:

- Establish network communication.
- Upload accumulated data.
- Receive remote commands.
- Return to sleep.

Separating measurement from communication often provides significant flexibility in managing the Battery Life Budget.

Communication Is Usually the Largest Energy Consumer

Compared with local processing, wireless communication typically consumes substantially more energy.

Communication activities may include:

- Network attachment
- Authentication
- Data transmission
- Acknowledgement reception
- Retry handling
- Network release

Depending on the communication technology and network conditions, these operations may dominate the energy consumed during each wake-up cycle.

For this reason, unnecessary communication should be avoided whenever operationally possible.

Batch Data Before Transmission

Rather than transmitting every measurement individually, firmware may accumulate multiple readings before initiating communication.

For example:

Measurement Frequency	Transmission Frequency
Hourly	Once per day
Every 15 minutes	Every 6 hours
Daily	Daily

This approach allows utilities to retain detailed consumption history while reducing the number of energy-intensive communication sessions.

The optimal balance depends on operational requirements, customer expectations, and regulatory obligations.

Schedule Around Operational Needs

Firmware schedules should reflect the business objectives of the utility rather than arbitrary technical intervals.

Examples include:

Daily Billing Applications

- One scheduled transmission per day may be sufficient.

District Metering

- More frequent reporting may support leakage monitoring.

Critical Infrastructure

- Alarm events may require immediate communication.

By aligning communication frequency with operational value, utilities avoid consuming battery energy on data that provides little additional benefit.

Separate Routine Data from Exception Events

Routine consumption reporting and exceptional operational events should not necessarily follow the same communication strategy.

Examples of **routine data** include:

- Consumption readings
- Battery status
- Temperature
- Diagnostic counters

Examples of **exception events** include:

- Leak detection
- Reverse flow
- Tamper alarm
- Burst pipe indication
- Communication fault

Routine information can often be scheduled.

Critical events may justify immediate transmission despite the additional battery cost.

This prioritisation allows battery energy to be allocated where it provides the greatest operational value.

Time Synchronisation Should Be Managed Carefully

Many communication technologies periodically synchronise device clocks with the network.

While accurate timekeeping is important for billing and analytics, excessive synchronisation may increase communication activity unnecessarily.

Firmware should therefore:

- Maintain local RTC accuracy where practical.
- Schedule synchronisation intelligently.
- Avoid unnecessary network interaction.

This helps preserve battery energy without compromising timestamp accuracy.

Design Schedules That Can Adapt

The optimal communication schedule may change throughout the product's operational life.

Examples include:

- Normal operation
- Drought management
- Leak investigation
- Emergency response
- Low battery mode

Firmware should therefore support configurable reporting schedules rather than fixed intervals wherever practical.

This flexibility allows utilities to respond to operational needs without replacing hardware.

Engineering Insight

The most energy-efficient communication is the communication that never needs to occur. Firmware should transmit information only when it provides meaningful operational value.

Practitioner Observation

In many AMI deployments, utilities initially request frequent reporting because the capability exists. After analysing operational data, reporting intervals are often relaxed because additional transmissions provide limited practical value while increasing battery consumption and network utilisation. Designing firmware with configurable scheduling allows communication behaviour to evolve with operational experience rather than being permanently fixed during product development.

Key Takeaways

- Meter reading and communication should be treated as separate firmware activities.
- Wireless communication is usually the largest energy consumer.
- Batching multiple readings into fewer communication sessions improves battery efficiency.
- Reporting frequency should be driven by operational objectives rather than technical capability.
- Flexible scheduling allows utilities to balance battery life with changing operational requirements.

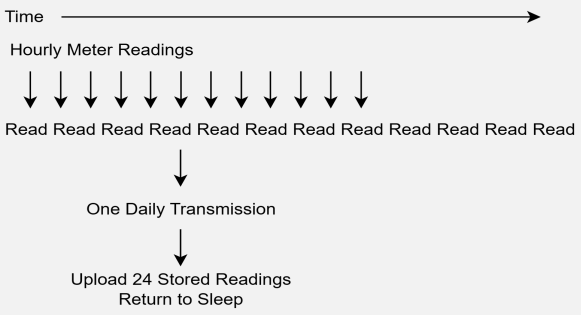

Figure 4.3 — Efficient Communication Scheduling

Separating meter reading from communication allows multiple measurements to be collected locally before a single scheduled transmission, reducing overall communication energy consumption.

4.4 Interrupt-Driven Firmware Versus Continuous Polling

One of the fundamental decisions in embedded firmware architecture is how the system determines when work needs to be performed.

Traditionally, many embedded applications rely on periodic polling, where the processor wakes at fixed intervals to check sensors, communication interfaces, or system status.

While polling is straightforward to implement, it often consumes energy even when no useful work is required.

For battery-powered smart water meters designed to operate for more than a decade, an interrupt-driven architecture generally provides a more efficient approach.

Rather than repeatedly asking whether something has happened, the processor remains asleep until the hardware indicates that action is required.

Understanding Polling

Polling involves checking system inputs at regular intervals regardless of whether any changes have occurred.

For example, firmware may wake every minute to:

- Read the pulse sensor
- Check tamper inputs
- Verify battery voltage
- Inspect communication status
- Review alarm conditions

After completing these checks, the processor returns to sleep.

If nothing has changed, the wake-up has consumed battery energy without producing operational value.

Over thousands of wake-up cycles each year, these unnecessary checks gradually reduce the available Battery Life Budget.

Understanding Interrupts

Interrupts reverse this operating model.

Instead of periodically checking every input, the processor remains in a low-power sleep state until notified by hardware.

Typical interrupt sources include:

- Pulse detection
- Real-Time Clock (RTC)

- Tamper switch activation
- Reverse flow detection
- Magnetic interference
- Communication module events
- External configuration commands

Only when an interrupt occurs does the firmware wake the processor, perform the required task, and immediately return to sleep.

This event-driven behaviour significantly reduces unnecessary processor activity.

Interrupts Reduce Both Energy and Latency

Interrupt-driven systems offer two important advantages.

First, they minimize battery consumption by eliminating unnecessary polling cycles.

Second, they often improve system responsiveness because the processor reacts immediately when an event occurs rather than waiting for the next polling interval.

For example, a tamper event detected by an interrupt can be processed instantly without requiring the firmware to wake periodically to inspect the tamper input.

Polling Still Has a Role

Although interrupt-driven design is generally preferable, polling has not disappeared from modern embedded systems.

Some functions naturally require periodic execution.

Examples include:

- Scheduled meter reading
- Battery health monitoring
- Daily reporting
- Clock synchronisation
- Self-diagnostic routines

The objective is therefore not to eliminate polling entirely, but to ensure that polling frequency is justified by operational requirements.

Where practical, multiple periodic activities should be combined into a single scheduled wake-up.

Avoid Interrupt Overload

Not every event should generate an interrupt.

Excessive interrupt activity may:

- Increase processor wake-ups
- Complicate firmware scheduling
- Increase power consumption
- Introduce timing conflicts
- Reduce system predictability

Firmware designers should therefore classify interrupts according to operational importance.

Examples include:

High Priority

- Meter pulse
- Tamper alarm
- Leak detection
- Brownout event
- Watchdog recovery

Medium Priority

- Communication completion
- RTC scheduled wake-up

Low Priority

- Diagnostic counters
- Maintenance statistics
- Development logging

Prioritising interrupts helps ensure that battery energy is reserved for events that provide meaningful operational value.

Design Firmware Around Events

Modern AMI firmware increasingly adopts an event-driven architecture.

Rather than continuously executing loops that inspect hardware status, firmware responds to:

- Hardware interrupts
- Communication events
- Scheduled timers
- Alarm conditions
- Remote commands

This architecture naturally supports lower energy consumption while simplifying future feature expansion.

Engineering Insight

Polling asks the same question repeatedly. Interrupts wait until the answer changes. Long-life battery-powered products benefit from listening rather than constantly asking.

Practitioner Observation

During firmware optimisation, engineers often focus on reducing processor execution time while overlooking unnecessary wake-up events created by periodic polling. In many long-life AMI products, migrating suitable functions from polling to interrupt-driven operation produces greater battery life improvements than optimising processor instruction efficiency.

Key Takeaways

- Interrupt-driven firmware reduces unnecessary processor wake-ups.
- Polling should be reserved for functions that genuinely require periodic execution.
- Event-driven architectures generally provide both lower energy consumption and faster response.
- Interrupt priority management improves both reliability and battery efficiency.
- Firmware architecture should minimise unnecessary system activity throughout the product lifecycle.

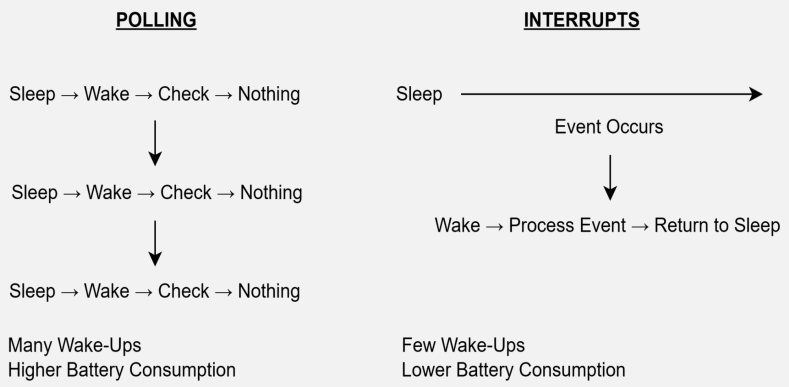

Figure 4.4 — Polling Versus Interrupt-Driven Operation

Interrupt-driven firmware minimises unnecessary wake-ups by allowing the processor to remain asleep until a meaningful event occurs, reducing energy consumption while maintaining rapid system response.

4.5 Communication Retry Strategy: The Hidden Battery Killer

Wireless communication is inherently probabilistic.

Unlike wired communication, successful transmission cannot always be guaranteed.

Signal strength, network congestion, environmental conditions, underground installations, and temporary network outages may all prevent data from reaching the utility backend.

To improve reliability, communication protocols typically employ retry mechanisms.

However, every retry consumes additional battery energy.

When retries become excessive, they can rapidly consume the Battery Life Budget while providing little improvement in operational performance.

Consequently, communication retry strategy should be regarded as a battery management function rather than simply a communication feature.

Retries Are Necessary—but Not Free

A retry is intended to recover from temporary communication failures.

Typical causes include:

- Poor cellular signal
- Temporary network congestion
- Base station maintenance
- Underground installations
- Basement meter chambers
- RF interference
- Packet collisions
- Server timeout

In these situations, retrying may successfully deliver the required data.

However, each retry repeats much of the communication process, including:

- Radio startup
- Network attachment (where required)
- Authentication
- Transmission
- Reception
- Waiting for acknowledgements

Each repetition consumes additional battery energy.

More Retries Do Not Always Improve Reliability

A common assumption is that increasing the retry count automatically improves communication success.

In reality, this relationship has diminishing returns.

For example:

Maximum Retry Count	Typical Result
1–2	Recovers most temporary failures
3–5	Small improvement in success rate
>5	Battery consumption increases significantly with limited additional benefit

If the underlying issue is persistent—such as poor RF coverage or a network outage—repeated retries simply consume battery energy without improving data availability.

The objective is therefore not to maximise retry attempts, but to optimise the balance between communication reliability and battery life.

Different Failure Types Require Different Responses

Not all communication failures are identical.

Firmware should distinguish between temporary and persistent failures.

Examples include:

Temporary Failures

- Network congestion
- Short-duration interference
- Server busy response
- Packet collision

These situations may justify an immediate retry.

Persistent Failures

- No network coverage
- Underground RF shadowing
- SIM provisioning issue
- Long-duration operator outage

Repeated retries are unlikely to succeed.

A more efficient strategy is to defer communication until the next scheduled reporting interval or after a defined backoff period.

Intelligent Backoff Conserves Battery

Rather than retrying immediately after every failure, firmware should progressively increase the waiting period before subsequent attempts.

A typical strategy may resemble:

- First failure → Retry after 1 minute
- Second failure → Retry after 15 minutes
- Third failure → Retry after 1 hour
- Continued failure → Resume at next scheduled reporting interval

This exponential backoff approach reduces unnecessary radio activity while still allowing recovery from temporary network issues.

It also reduces network congestion by preventing large numbers of devices from repeatedly transmitting at the same time.

Preserve Data Rather Than Repeating Measurements

Communication failure should not require repeating the meter reading.

Instead, firmware should:

- Store validated readings locally.
- Preserve timestamps.
- Queue unsent records.
- Upload accumulated data when communication becomes available.

This approach ensures that temporary communication failures do not result in data loss or unnecessary sensor activity.

The objective is to retry the transmission—not to recreate the measurement.

Retry Strategy Must Match Utility KPIs

Firmware retry behaviour should support the operational objectives defined by the utility.

For example:

Daily Billing

A delayed transmission of several hours may still satisfy operational requirements if the daily reading is successfully delivered.

Leak Detection

Critical alarms may justify additional retry attempts because timely notification provides operational value.

Routine Diagnostics

Repeated retries are often unnecessary if the information can be delivered during the next scheduled upload.

Battery energy should therefore be allocated according to the importance of the information being transmitted.

Engineering Insight

The most efficient retry is the one that has the highest probability of success. Repeating unsuccessful transmissions without understanding the cause simply converts battery energy into radio activity.

Practitioner Observation

In field deployments, poor communication performance is often addressed by increasing the maximum retry count. While this may slightly improve short-term delivery rates, it can also accelerate battery depletion without resolving the underlying RF coverage problem. A better approach is to investigate why retries are occurring and optimize network conditions, antenna performance, installation practices, or reporting schedules rather than relying solely on additional transmission attempts.

Key Takeaways

- Every retry consumes additional battery energy.
- Increasing retry count does not necessarily improve communication reliability.
- Firmware should distinguish between temporary and persistent failures.
- Intelligent backoff strategies reduce unnecessary battery consumption.
- Retry behaviour should be aligned with utility operational priorities and data availability KPIs.

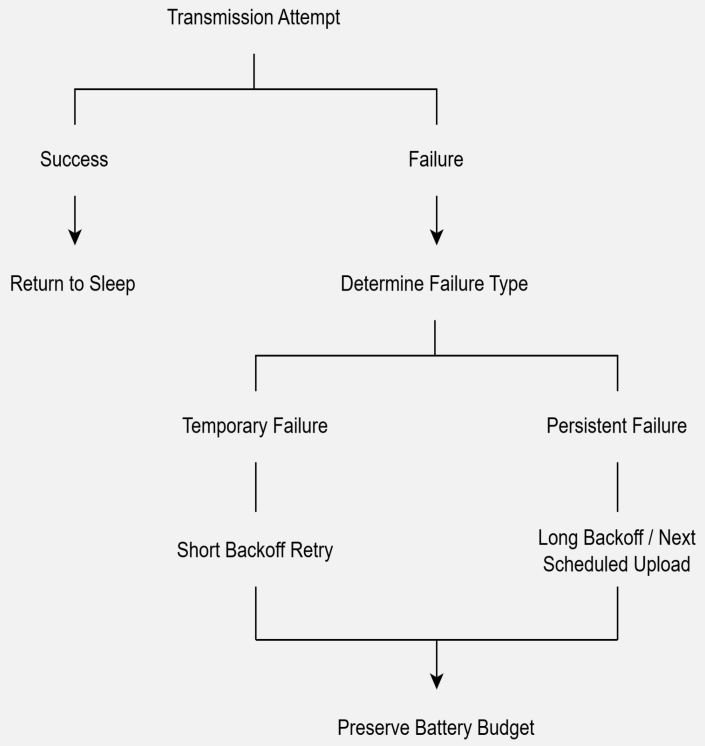

Figure 4.5 — Intelligent Communication Retry Strategy
Intelligent retry strategies classify communication failures and apply appropriate recovery behaviour, improving battery efficiency while maintaining reliable data delivery.

4.6 OTA Firmware Updates Without Destroying Battery Life

Modern smart water meters are expected to remain in service for ten to fifteen years.

During this period, firmware rarely remains unchanged.

Utilities may require:

- Bug fixes
- Cybersecurity patches
- Communication protocol updates
- Performance improvements
- Regulatory compliance updates
- New operational features

Replacing thousands of deployed meters simply to update firmware is neither practical nor economical.

Over-the-Air (OTA) firmware updating has therefore become an essential capability for modern AMI systems.

However, OTA is one of the most energy-intensive operations performed by a battery-powered smart meter.

Without careful firmware design, a single update campaign can consume a significant portion of the Battery Life Budget.

OTA Is Not Routine Communication

Routine data reporting typically involves transmitting only a small amount of information.

Examples include:

- Consumption data
- Battery status
- Alarm events

Firmware updates are fundamentally different.

They require:

- Downloading a much larger data payload
- Verifying data integrity
- Writing to Flash memory
- Managing firmware images
- Rebooting the device
- Validating successful installation

Each stage consumes considerably more energy than normal reporting.

OTA should therefore be regarded as a special maintenance activity rather than routine communication.

Design OTA for Reliability First

An interrupted firmware update can render a smart meter inoperable if recovery mechanisms are not available.

Firmware should therefore support:

- Image integrity verification
- Secure authentication
- Resume capability after interruption
- Automatic rollback
- Recovery after power loss
- Version management

Reliable recovery is more important than update speed.

A firmware update that succeeds on the second attempt is preferable to one that permanently disables the device.

Schedule OTA Intelligently

Firmware updates should not occur immediately whenever a new version becomes available.

Instead, firmware should evaluate whether update conditions are suitable.

Examples include:

- Adequate battery voltage
- Stable network conditions
- Acceptable signal strength
- Sufficient available memory
- Appropriate maintenance window
- Utility approval

Postponing an update under poor operating conditions often conserves more battery energy than repeatedly attempting unsuccessful downloads.

Avoid Updating Every Device Simultaneously

Large AMI deployments may contain tens or hundreds of thousands of meters.

Updating every device simultaneously may create:

- Network congestion
- Backend overload
- Increased retry activity
- Reduced data availability

Firmware should therefore support staged deployment strategies.

Examples include:

- Geographic rollout
- Device groups
- Pilot deployments
- Percentage-based rollout
- Utility-defined maintenance batches

This approach improves both operational control and battery efficiency.

Protect Metering During OTA

Firmware updates should never compromise the primary function of the smart water meter.

During the update process, the system should continue to preserve:

- Meter index
- Consumption history
- Calibration constants
- Configuration parameters
- Event logs

Where practical, essential metering functions should remain protected even if communication services are temporarily suspended during the update.

Design OTA With the Battery Life Budget in Mind

OTA updates should be treated as planned investments of battery energy.

Engineers should estimate:

- Download energy
- Flash programming energy
- Verification energy
- Reboot energy
- Recovery energy
- Retry allowance

These activities should be incorporated into the Battery Life Budget rather than assumed to be insignificant.

For example, a product expected to receive three firmware updates during its lifetime should reserve sufficient battery capacity for those updates during the original design phase.

Secure OTA Without Excessive Energy Consumption

Security mechanisms such as:

- Digital signatures
- Image authentication
- Encryption
- Secure boot
- Integrity verification

are essential for protecting critical infrastructure.

However, they also require processor execution time and memory access.

Firmware should therefore implement security efficiently, balancing cybersecurity requirements with battery constraints.

Security should never be omitted to save energy, but neither should it be implemented inefficiently.

Engineering Insight

OTA is not a software feature—it is a lifetime maintenance strategy. The battery energy required for future firmware updates should be considered during product design, not after deployment.

Practitioner Observation

In many AMI projects, battery life calculations assume normal reporting behaviour but overlook firmware maintenance throughout the product's lifetime. In reality, multiple firmware updates may be required over ten to fifteen years to address cybersecurity, network evolution, or operational improvements. Reserving battery capacity for these planned maintenance activities during the original Battery Life Budget calculation helps prevent unexpected reductions in service life.

Key Takeaways

- OTA updates are among the most energy-intensive firmware operations.
- Reliability is more important than update speed.
- Firmware updates should be scheduled only under suitable operating conditions.
- Large deployments benefit from staged rollout strategies.
- The Battery Life Budget should include energy reserved for future firmware maintenance.

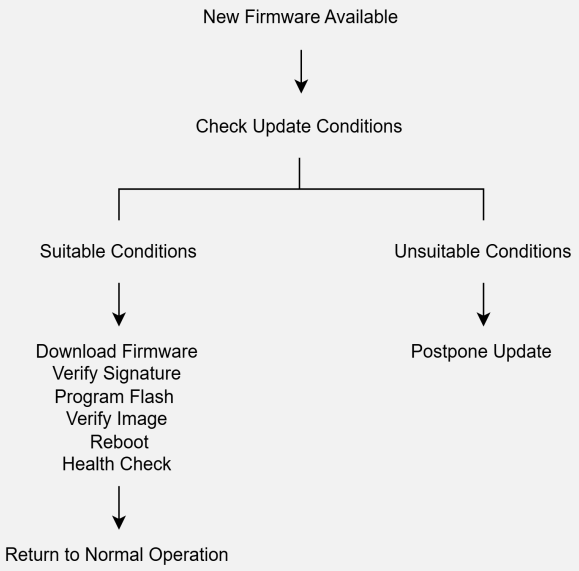

Figure 4.6 — Energy-Aware OTA Firmware Update Workflow

OTA updates should proceed only when operating conditions are suitable, ensuring reliable firmware maintenance while preserving battery energy and system integrity.

4.7 Battery-Aware Firmware Design

Ultra-low-power firmware should not treat battery energy as an unlimited resource.

Instead, every firmware decision should recognise that the available battery capacity must support the entire operational lifetime of the smart water meter.

This requires firmware to become aware of the battery's condition and adapt its behaviour accordingly.

Rather than operating with fixed assumptions throughout the product's life, battery-aware firmware dynamically adjusts system behaviour to maximise operational value while preserving the remaining Battery Life Budget.

Battery Awareness Goes Beyond Voltage Monitoring

Battery-aware firmware is often misunderstood as simply measuring battery voltage.

In reality, battery awareness considers multiple factors, including:

- Battery voltage
- Estimated remaining capacity
- Temperature
- Communication success rate
- Reporting history
- Device age
- Operational mode
- Recent power consumption trends

Considering these factors together allows firmware to make better decisions than relying on voltage alone.

Adapt System Behaviour as the Battery Ages

The operating strategy of a newly deployed meter does not necessarily remain appropriate after ten years of service.

As available battery energy decreases, firmware may progressively adjust system behaviour.

Examples include:

Normal Operation	Low Battery Mode
Full diagnostics	Essential diagnostics only
Frequent status reports	Reduced reporting frequency
Background maintenance	Deferred where practical
Extended logging	Critical events only
Full communication capability	Prioritised communication

The objective is to preserve the functions that deliver the greatest operational value.

Prioritise Essential Functions

Not all firmware activities contribute equally to utility operations.

Battery-aware firmware should classify system functions according to operational importance.

Essential Functions

- Meter reading
- Billing data
- Critical alarms
- Secure operation
- Configuration integrity

Important Functions

- Diagnostic reporting
- Performance statistics
- Maintenance information

Optional Functions

- Extended debug logging
- Development telemetry
- Non-essential analytics

When battery capacity becomes limited, optional activities should be reduced before essential metering functions are affected.

Allocate Energy According to Business Value

Battery energy should be invested where it delivers measurable operational benefit.

For example:

Increasing transmission frequency from one report per day to one report per hour may increase battery consumption significantly.

Before implementing such a feature, engineers should ask:

- Does this additional data improve billing?
- Does it improve leak detection?
- Does it reduce operational cost?
- Does it support regulatory compliance?

If the answer is no, the additional battery consumption may not be justified.

Monitor Long-Term Energy Trends

Battery-aware firmware should observe long-term behaviour rather than reacting only to instantaneous measurements.

Useful indicators include:

- Average communication retries
- Reporting success rate
- Average transmission duration
- Battery voltage trend
- Temperature history
- Alarm frequency
- Unexpected wake-up count

These trends allow firmware and utility operators to identify gradual degradation before it develops into operational problems.

Design for Graceful Degradation

As battery energy becomes limited, firmware should continue delivering useful service rather than operating normally until sudden failure occurs.

A typical degradation strategy may include:

Stage 1 – Normal Operation

- Full functionality
- Scheduled reporting
- Complete diagnostics

↓

Stage 2 – Conservation Mode

- Reduced background activity
- Lower communication frequency
- Optional diagnostics suspended

↓

Stage 3 – Critical Mode

- Meter reading maintained
- Critical alarms preserved
- Essential communication only

↓

Stage 4 – End-of-Life

- Final low battery notification
- Preserve meter data
- Controlled shutdown

This staged approach maximises operational value throughout the remaining service life.

Firmware Should Be Measured Against the Battery Life Budget

Every significant firmware enhancement should answer the same engineering questions:

- How much additional energy will this feature consume?
- How often will it execute?
- How many additional wake-ups will it create?
- Does it affect communication frequency?
- Does it reduce battery life?
- Does the operational benefit justify the additional energy?

These questions transform battery life from an afterthought into a formal engineering requirement.

Engineering Insight

> **Battery-aware firmware continuously asks one question: "Is this operation worth the energy it will consume?" Long-life products are built by answering that question correctly thousands of times over their operational lifetime.**

Practitioner Observation

In mature AMI deployments, firmware rarely remains static. New utility requirements, cybersecurity updates, analytics features, and regulatory changes gradually expand system functionality. Without disciplined energy management, these incremental additions can slowly consume the battery margin originally reserved during product

design. Battery-aware firmware helps ensure that feature growth remains aligned with the original Battery Life Budget rather than unintentionally reducing service life.

Key Takeaways

- Battery-aware firmware adapts system behaviour throughout the product lifecycle.
- Battery management involves more than simply measuring voltage.
- Essential utility functions should always receive the highest energy priority.
- Long-term operational trends provide better guidance than individual measurements.
- Every firmware enhancement should be evaluated against its impact on the Battery Life Budget.

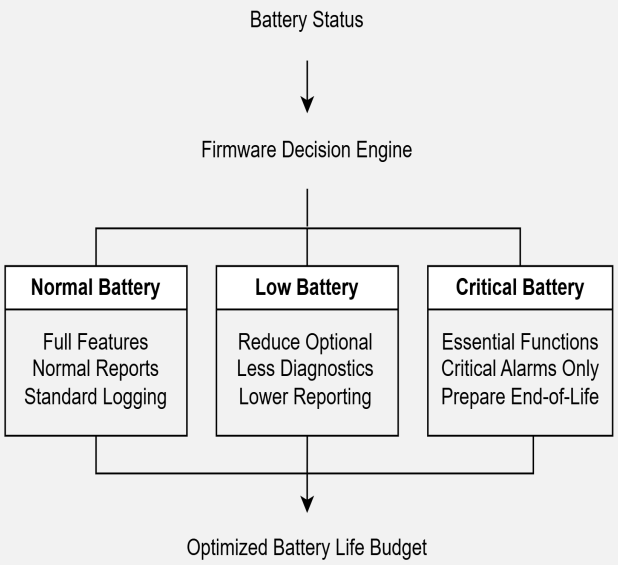

Figure 4.7 — Battery-Aware Firmware Decision Framework
Battery-aware firmware continuously evaluates battery condition and adjusts system behaviour to preserve essential utility functions while optimising the remaining Battery Life Budget.

4.8 Firmware Optimisation Is Never Finished

Unlike hardware, which becomes largely fixed once a product enters production, firmware continues to evolve throughout the operational life of a smart water meter.

Over a deployment lasting ten to fifteen years, firmware may undergo multiple revisions to address:

- Software defects
- Cybersecurity vulnerabilities
- Communication network evolution
- Regulatory requirements
- Utility operational needs
- Performance improvements
- New product capabilities

Each revision has the potential to change battery consumption.

Consequently, battery life should not be regarded as a characteristic established during product development and then assumed to remain unchanged throughout deployment.

Firmware optimization is a continuous engineering process.

New Features Always Have an Energy Cost

As products mature, firmware naturally becomes more capable.

Utilities may request:

- Additional alarms
- More detailed diagnostic information
- Higher reporting frequency
- Expanded data logging
- Enhanced cybersecurity
- New communication protocols

While each enhancement may appear beneficial in isolation, together they increase processor activity, memory access, communication overhead, and system wake-ups.

Without disciplined review, gradual feature expansion can consume the battery margin originally reserved during product design.

Battery Life Validation Should Continue After Release

Many organisations perform battery life validation only before product release.

However, firmware updates introduced years later may significantly alter energy consumption.

Whenever major firmware changes are introduced, engineering teams should reassess:

- Sleep current
- Wake-up frequency
- Active processing time
- Communication duration
- Retry behaviour
- Battery monitoring frequency
- OTA energy consumption

This ensures that long-term battery life objectives remain achievable.

Measure Rather Than Assume

Battery optimisation should be based on measurement rather than expectation.

Useful firmware performance indicators include:

- Average daily wake-ups
- Active time per wake-up
- Communication success rate
- Retry frequency
- Average transmission duration
- Sleep current verification
- Daily energy consumption estimate

Tracking these metrics across firmware versions allows engineering teams to detect unintended increases in energy consumption before they affect field deployments.

Include Battery Impact in Every Firmware Review

Battery life should become a standard consideration during firmware design reviews.

Alongside questions relating to functionality, reliability, and cybersecurity, reviewers should also ask:

- Does this feature introduce additional wake-ups?
- Does it increase communication activity?
- Does it require additional Flash writes?
- Does it affect sleep duration?
- Can the same objective be achieved more efficiently?

Embedding these questions into the development process helps preserve battery life throughout the product lifecycle.

Collaboration Across Engineering Disciplines

Firmware optimization cannot be performed in isolation.

Successful long-life AMI products require collaboration between:

- Hardware engineers
- Firmware developers
- RF specialists
- Test engineers
- Quality engineers
- System architects
- Utility operators

Changes in one area often influence battery performance elsewhere.

For example, a network configuration change may increase communication retries, while a hardware modification may reduce wake-up time.

Maintaining battery performance therefore requires a systems engineering perspective.

Continuous Improvement Benefits Utilities

Firmware optimization does more than extend battery life.

It also contributes to:

- Improved data availability
- More predictable maintenance planning
- Reduced field visits
- Better customer service
- Lower lifecycle costs
- Increased confidence in long-term deployments

Battery optimisation should therefore be viewed as part of overall utility operational excellence rather than purely a technical objective.

Engineering Insight

> **The best battery optimization is not a one-time achievement. It is a development culture where every firmware revision is evaluated for its impact on long-term energy consumption.**

Practitioner Observation

Experienced AMI development teams often discover that battery life changes gradually rather than suddenly. A few additional diagnostic messages, a revised communication protocol, or more frequent health checks may appear insignificant individually, but together they can reduce battery life over several firmware releases. Maintaining an energy performance baseline for every production firmware version helps ensure that continuous feature development does not unintentionally erode long-term product performance.

Key Takeaways

- Firmware optimization continues throughout the product lifecycle.
- Every firmware enhancement should be evaluated for its battery impact.
- Battery validation should accompany major firmware releases.

- Energy consumption should be measured using objective performance metrics.
- Long-term battery performance depends on collaboration across multiple engineering disciplines.

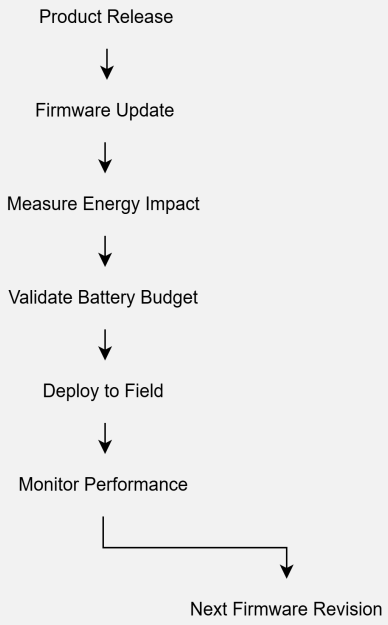

Figure 4.8 — Continuous Firmware Optimisation Lifecycle
Battery optimisation is an ongoing lifecycle activity. Each firmware revision should be evaluated against the original Battery Life Budget to ensure that long-term service life objectives remain achievable.

Summary

Firmware is the operational intelligence that determines how battery energy is consumed throughout the life of a smart water meter.

While hardware establishes the foundation for low-power operation, firmware decides when the processor wakes, how long it remains active, when communication occurs, how failures are handled, and how the remaining Battery Life Budget is allocated.

This chapter has shown that long battery life is achieved not through a single optimisation, but through disciplined firmware architecture. Efficient sleep strategies, intelligent scheduling, event-driven processing, adaptive retry mechanisms, energy-aware OTA updates, and battery-aware decision making all contribute to extending operational life while maintaining reliable utility performance.

Importantly, firmware optimization does not end when a product is released. As AMI systems evolve over many years, maintaining battery performance requires continuous measurement, validation, and engineering discipline across every firmware revision.

The next chapter shifts from design to validation. It examines how laboratory testing, environmental qualification, accelerated ageing, and field verification are used to confirm that theoretical battery life predictions can be achieved under real-world deployment conditions.

Disclaimer

The views and observations expressed in this paper are the author's own and are intended for general industry discussion and educational purposes. They are based on publicly available research, practical engineering experience, and regional industry observations, and do not represent the official position, product commitments, customer relationships, or commercial practices of any employer, client, utility, or affiliated organisation. Utilities should evaluate deployment decisions within their own operational, technical, regulatory, and procurement context.

About the Author

Vincent Wong (Wong Yew Hon)

Vincent Wong brings over **25 years of engineering and product leadership experience** across industrial infrastructure and technology sectors, with the past decade focused on smart metering, Advanced Metering Infrastructure (AMI), and NB-IoT deployment strategies in ASEAN utility environments. As a senior engineering and product development executive, his work spans hardware engineering, connectivity systems, and utility digitalization initiatives. His writing focuses on the practical operational, connectivity, deployment, and governance realities of smart utility programs in Southeast Asia.

Connect on LinkedIn: linkedin.com/in/vincentwongmy