# AN14859

## How to Leverage Generative AI to Accelerate Software Development on NXP MCX MCUs

Rev. 1.0 — 5 November 2025 Application note

**Document information**

| Information | Content |
|---|---|
| Keywords | AN14859, MCX, GenAI MCU software development |
| Abstract | This application note describes how to leverage generative AI to accelerate software development on NXP MCX MCUs. |

# 1 Introduction

This application note explains the integration of generative AI tools in NXP MCU software development workflows. As AI large language models have experienced exponential growth in recent years, AI-assisted programming has become increasingly prevalent across the software development industry. The embedded systems development community can benefit significantly from understanding and adopting these powerful tools to enhance productivity and code quality.

Rather than viewing AI as a replacement for traditional development practices, this document positions AI as a complementary tool that can accelerate common development tasks while maintaining the critical role of human expertise in embedded systems design.

## 1.1 Understanding generative AI from an MCU developer's perspective

Generative AI can be conceptualized as an intelligent programming assistant with the following characteristics:

- Extensive knowledge base covering programming languages, development frameworks, and embedded systems best practices
- 24/7 availability for code generation, problem-solving, and technical guidance
- Ability to understand context and generate targeted solutions

The traditional development process is as follows:

1. Encounter a problem
2. Search online resources
3. See technical documentation and reference example code
4. Modify and adapt the solution

With AI, the process is as follows:

1. Encounter a problem
2. Ask the AI directly
3. The AI generates targeted code and explanations
4. Test and verify → complete development

For example, in the traditional approach, you must write an SPI driver for a new MCU:

1. Open the chip reference manual (may be 1000+ pages)
2. Locate the SPI chapter
3. Understand register configuration and reference examples
4. Study and comprehend the SDK’s SPI example code
5. Example code analysis → custom implementation → debugging

Use the AI as follows:

- You: Help me write an SPI1 driver for NXP MCXA346 to read/write an external SPI Flash chip, Flash model is "xxx", hardware connection is "xxxx."
- AI: Sure, let me generate a complete SPI driver code for you (directly provides usable code framework including initialization, read/write functions, and so on).

For those who have never used generative AI (referred to as AI) for programming assistance, this may sound exciting. However, developers must understand its limitations:

- AI-generated code cannot guarantee compilation success or functional correctness. All AI outputs require human review and validation, particularly in embedded systems where hardware constraints are critical.

- Embedded applications are highly dependent on specific hardware configurations. AI lacks inherent knowledge of particular MCU variants, pin configurations, clock settings, and resource constraints, unless explicitly provided.
- AI cannot replace hardware-in-the-loop testing. Generated embedded code requires validation on actual target hardware with appropriate test scenarios.

In summary: AI functions as a highly knowledgeable but fallible programming assistant. When properly used, it can significantly improve development efficiency, but the final code quality assurance is the responsibility of the human developer.

# 2 AI model landscape and tool overview

This section describes the AI model landscape and tool.

## 2.1 Leading AI models for programming applications

Based on practical evaluation in embedded development scenarios, the following models demonstrate superior performance:

1. Claude 4.5 Sonnet (Anthropic): Next-generation hybrid reasoning model with enhanced autonomous operation, tool utilization, and complex task-handling capabilities. It provides superior code generation and multi-step reasoning with extensive context support.
2. Claude Opus (Anthropic): Currently the most capable model for complex problem solving, suitable for advanced AI agent systems.
3. GPT-5 (OpenAI): Advanced model with significant improvements in reasoning capabilities and processing speed.
4. Grok 4(xAI): xAI's flagship enterprise-grade AI model featuring multi-agent reasoning architecture and PhD-level professional capabilities across all domains.

Additional models, such as DeepSeek, Llama, and Kimi, also offer excellent capabilities. However, based on practical experience in MCU development scenarios, the aforementioned four models demonstrate superior performance. The AI model capabilities evolve rapidly and developers must monitor performance benchmarks and adjust the tool selection accordingly.

Third-party evaluation platforms, such as www.datalearner.com, provide current rankings based on various testing standards, including HumanEval and MBPP benchmarks.

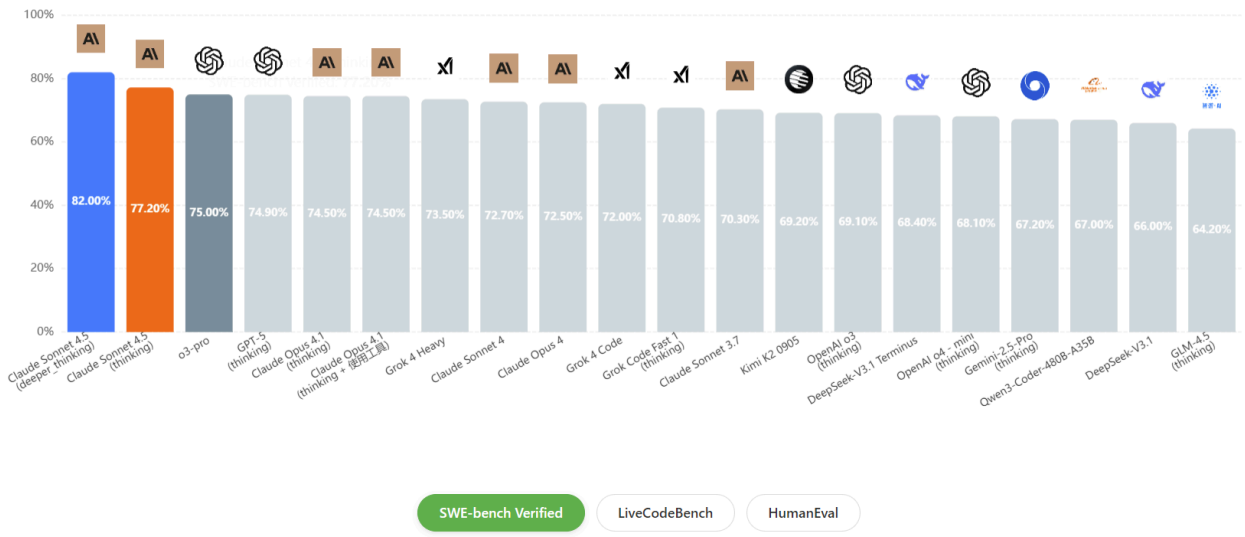


**Figure 1. LLM programming capability rankings (SWE-bench, September 2025) from datalearner.com**

**Agents: AI agents: autonomous development assistants**

The AI models described above serve as "engines" providing code understanding and generation capabilities. However, when dealing with complex programming environments, using them in isolation can be inefficient. Traditional workflows require manual code copying, pasting, compilation checking, and iterative refinement. AI agents address this limitation by providing autonomous task-execution capabilities.

Traditional approach:

1. Developer requests: "Help me write an SPI driver"
2. AI provides the code
3. Developer encounters a compilation error
4. AI modifies the code
5. Issues persist, requiring multiple iterations

Agent approach:

1. Developer: "Help me write an SPI driver and ensure it compiles successfully"
2. Agent:
   a. Checks the project structure and MCU model
   b. Generates the code
   c. Attempts a compilation - missing header file detected
   d. Automatically adds the includes
   e. Compiles again successfully
   f. Writes the test function as a bonus

## 2.2 Mainstream AI-assisted programming tools

Current AI-assisted programming tools have evolved beyond simple conversational interfaces, offering two primary implementation approaches:

- VS Code plugin-based:
  - Examples: GitHub Copilot, Cody, Continue, Augment

  - Advantages: Preserves your VS Code setup, key bindings, and plugins; runs multiple assistants side by side and switches on demand
- Independent IDEs (VS Code derivatives):
  - Examples: Cursor, TRAE
  - Advantages: Deeply integrated AI experiences with tailored UI and workflows beyond a plugin

### 2.3 How to choose a model

Current AI-assisted programming tools primarily support the VS Code ecosystem and have not yet integrated directly with traditional embedded IDEs, such as MCUXpresso, Keil, or IAR. This reflects the reality that general-purpose programming tools often advance faster than specialized domain tools.

- Option A: VS Code's robust plugin system now supports virtually all NXP MCU programming IDE functionalities. NXP's official MCUXpresso for VS Code plugin ports the MCUXpresso IDE capabilities to VS Code, providing the editing, compilation, download, and debugging functionality.
- Option B: Continues using existing IDEs while using VS Code as an AI-enhanced "super editor" for code generation and analysis tasks.
- Option C: Skips VS Code altogether. Uses standalone applications (Monica, Kimi) or web-based AI platforms for conversational programming assistance without IDE integration.

## 3 Practical experience - NXP FRDM-MCXA346 with VS Code Copilot

This section demonstrates AI-assisted programming using the NXP FRDM-MCXA346 hardware platform and VS Code with the GitHub Copilot plugin.

***Note:*** *This tutorial is IDE-agnostic. Whether using Keil, IAR, or MCUXpresso, the concepts apply. VS Code serves as an AI-enhanced "super editor" while compilation, download, and debugging can use familiar toolchains.*

### 3.1 Prerequisites

- Hardware platform: FRDM-MCXA346
- Development environment: VS Code + GitHub Copilot extension
- SDK: See the *Getting Started with FRDM-MCXA346* guide for SDK download and setup instructions: https://www.nxp.com/document/guide/getting-started-with-frdm-mcxa346:GS-FRDM-MCXA346
- Make sure that you have downloaded the FRDM-MCXA346 SDK and ran the Hello World example.
- Make sure that you have VS Code installed and that you are familiar with basic VS Code operations.

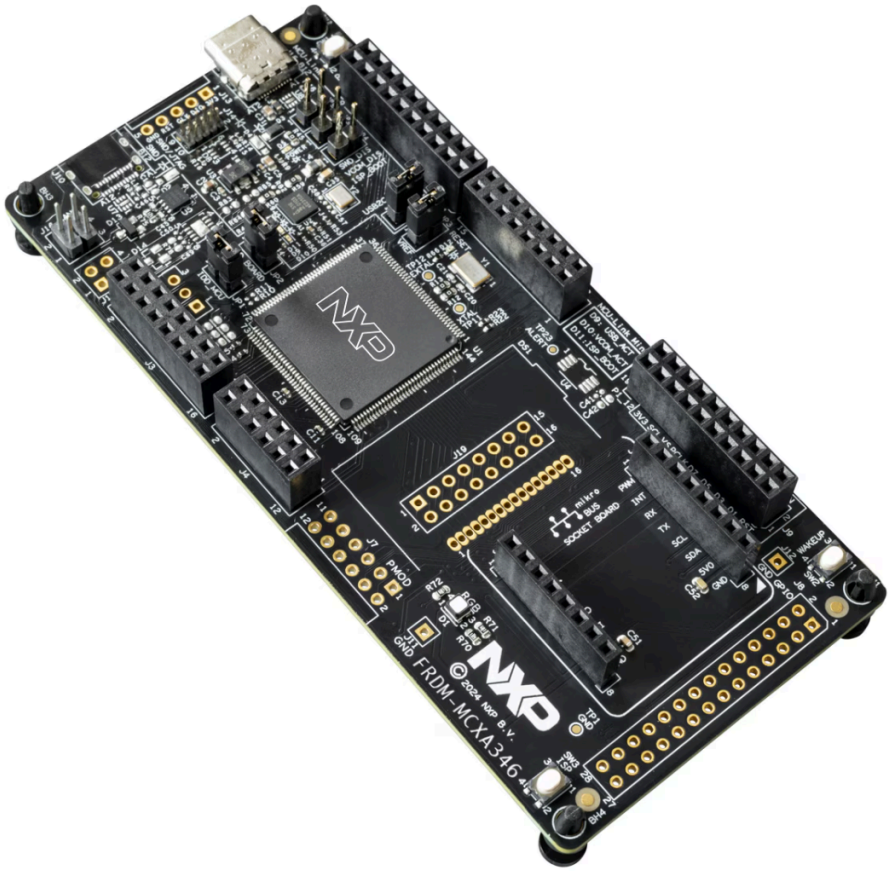


Figure 2. FRDM-MCXA346

## 3.2 Installing Copilot extension

Install the GitHub Copilot and GitHub Copilot Chat extensions from the VS Code marketplace.

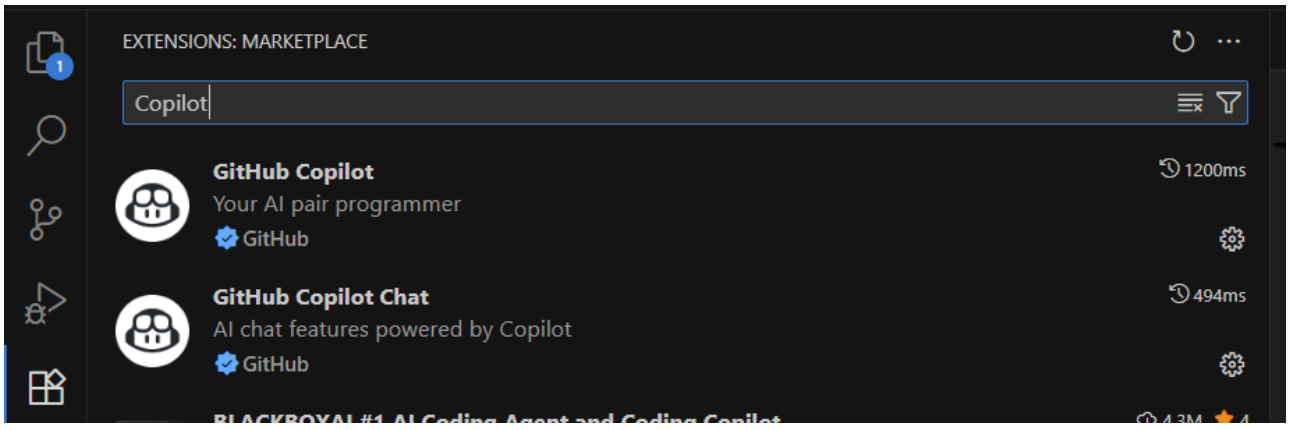


Figure 3. Installing Copilot extension

## 3.3 Importing project into VS code

Drag the MCUXpresso Config Tool-generated project source directory into VS Code and use Ctrl+Alt+I to launch the Copilot chat interface.

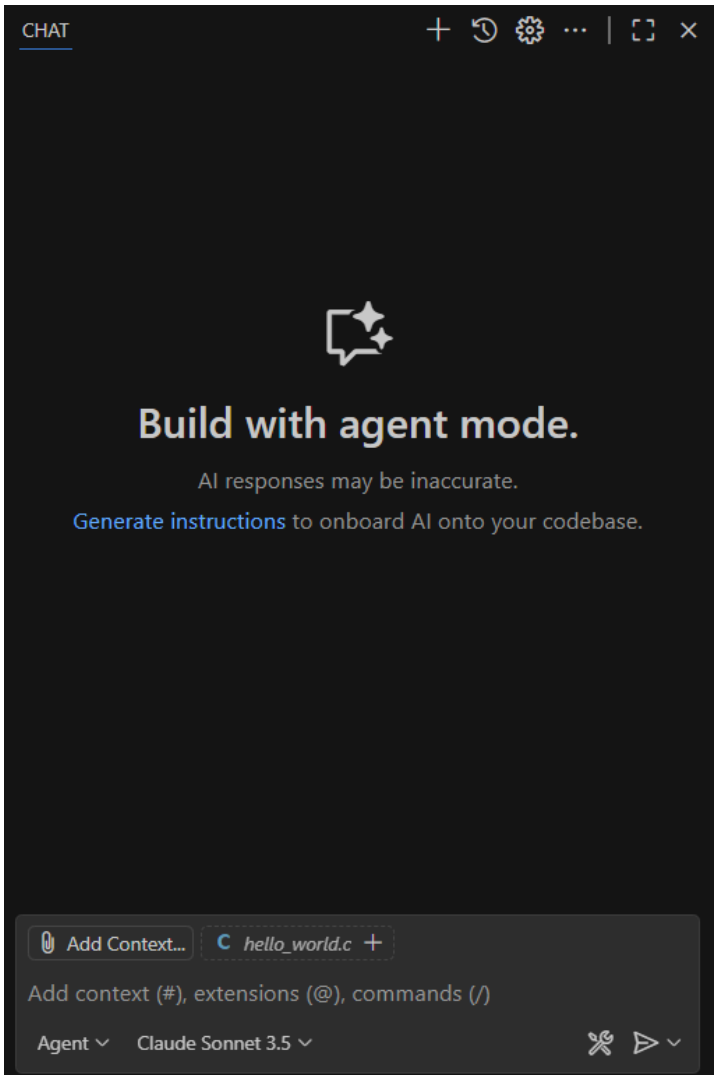


**Figure 4. Copilot chat panel**

## 3.4 Simple trial

Ask a concrete task in the default Agent mode and Copilot produces a complete answer.

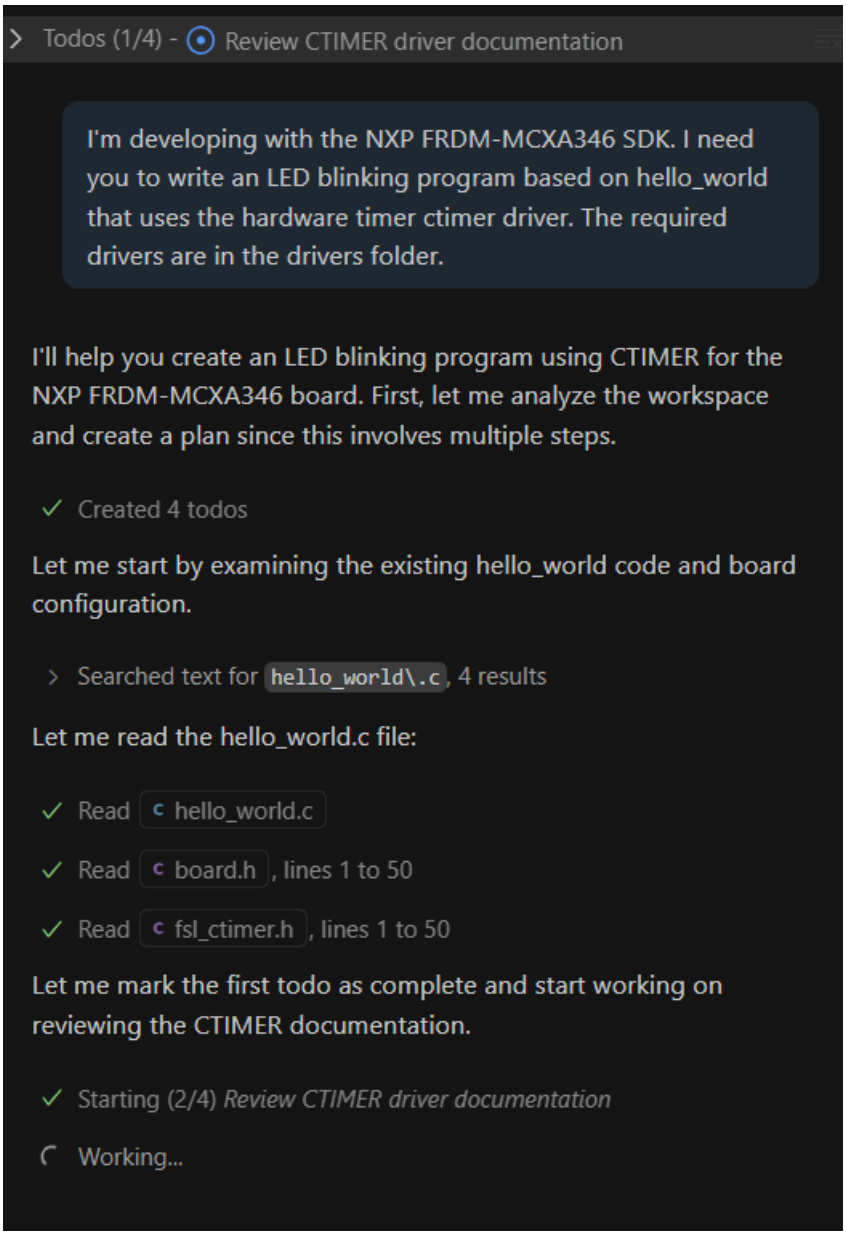


**Figure 5. Ask Copilot a question in Agent mode**

While you should not expect a perfect solution generated by AI, but the solution generated by AI may only need minor review and edits to get complied successfully, which saves considerable development time.

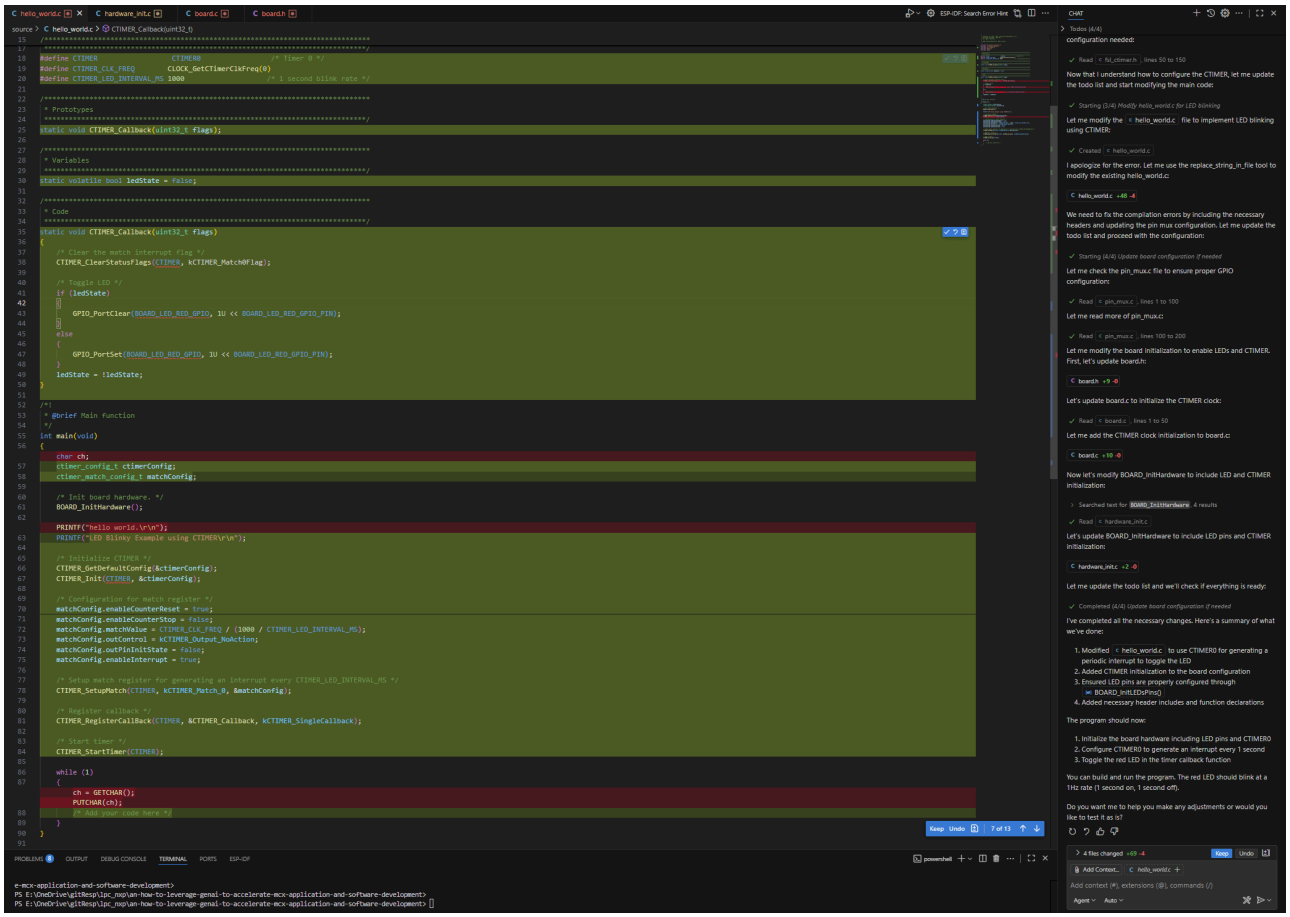

**Figure 6. In Agent mode, Copilot analyzes the problem and modifies the code directly**

Copilot can work in several modes:

- Chat: conversational Q&A; good for explanations and brainstorming
- Agent: autonomously tackles multi-step tasks and can modify local files if allowed
- Edit: AI-assisted edits within files
- Inline: real-time code completion beyond standard IntelliSense

For details, refer to the official Copilot tutorials: https://github.com/features/copilot/tutorials

In the following sections, we cover practical usage patterns and techniques to get the most out of AI-assisted MCU development.

# 4 Best practices

Based on the experience using AI in embedded development, there are several practical guidelines.

## 4.1 Iterative validation methodology: applying circuit debugging principles to AI-assisted software development

Avoid expecting AI to provide perfect solutions in a single interaction. Like debugging complex circuits, systematic step-by-step validation produces better results.

**Incorrect approach**: Staring at the whole schematic trying to identify all problems simultaneously.

**Correct approach**: Using oscilloscope to test level by level, starting from the power supply, validating each module systematically.

AI-assisted programming follows the same principle. Do not expect the AI to generate a 100 % correct solution on the first try, that is not realistic. Iterate step by step.

**Example**:

Implementing UART and DMA for variable-length data reception is a common task for embedded system design, which is needed in nearly every product that uses serial communication. If you ask the AI to do this all at once, the result is often not optimal.

- You: "I need UART and DMA variable-length reception for FRDM-MCXA346."
- AI: Provides large codeblock with potential bugs
- You: Test it, it does not work
- AI: Uncertain about errors, provides a different large-codeblock
- Result: Circular debugging, wasted time

Effective iterative approach:

1. Basic UART initialization and data transmission/reception:
   - AI provides code → developer tests it → functional
   - AI learns the hardware platform and programming style
2. UART interrupt functionality, especially for RX idle line features:
   - AI modifies code based on step 1 → developer tests it → functional
   - AI understands the interrupt-handling approach
3. Adding DMA integration:
   - AI continues building on the previous code → developer tests it → functional
   - Each step builds on the validated foundation

If you decompose the task and validate each output incrementally, the success rate is higher than asking the AI to do everything at once.

A good decomposition strategy should:

- Implement core functionality first, then add peripheral features
- Each step has clear pass/fail criteria
- Keep code changes per a step moderate in size
- Let the AI modify the existing code rather than rewrite it from scratch

**Why does it work?**

- Complexity reduction: Complex problems are combinations of simple problems. The AI success rate with simple problems far exceeds the rate for complex problems.
- Context building: Each successful code iteration becomes a "reference answer" for the subsequent requests. The AI gradually understands programming habits and project characteristics.
- Rapid problem isolation: When errors occur, immediate problem identification is possible without searching through extensive codeblocks.

## 4.2 Leveraging AI to discover existing solutions

Professional software engineers use proven, well-tested, existing code rather than implementing every function from scratch. This concentrates effort on solving previously unsolved problems rather than low-level repetition.

**Example**:

Let's suppose that your MCU project must receive and parse user commands via UART:

**Traditional approach**:

- You ask the AI: "Help me write a serial command-parsing function"
- AI provides a basic string-processing implementation with potential bugs

**Discovery-oriented approach**:

- "What mature libraries or standard implementations exist for MCU command-line parsing?"
- AI response:
  - microrl library (MCU-specific command-line library)
  - FreeRTOS and CLI component
  - Lightweight getopt port
  - AT command parser standard patterns
  - Embedded line noise port

This approach discovers mature, battle-tested components rather than custom implementations.

**Why does it work?**

- Robust code: Extensively validated with comprehensive-edge case handling
- Reduced maintenance: Standard interfaces with a rich documentation facilitate future extensions

By using the AI to discover existing solutions, you not only save development time but also gain higher-quality and more secure solutions.

## 4.3 Importance of prompt engineering and context

The prompt quality directly determines the AI output quality. Clear task descriptions and comprehensive background information enable the AI to generate more suitable code.

**Example**:

Let's suppose that you want to implement button debouncing for your product. See how different prompts lead to different outcomes.

**Bad prompt**: "Help me write button-debouncing code"

AI may provide generic, impractical framework without platform specifics, polling vs. interrupt approach, debounce timing, or multi-button support.

**Effective prompt**: "I need button-debouncing functionality for the NXP MCXA346 platform with specific requirements:

1. Hardware: Three independent buttons connected to GPIO P1_0, P1_1, P1_2, active low
2. Method: Timer-polling approach, no interrupts
3. Debounce time: 20 ms
4. Functionality: Detecting press, release, long-press (two seconds) events
5. Interface: Callback mechanism for application-layer event registration
6. Code style: State machine implementation with detailed comments
7. Resource constraints: Minimizing RAM usage due to project limitations"

**Result**: AI generates usable code

A good prompt specifies all constraints precisely. Just as designing hardware circuits requires clear electrical parameters, writing AI prompts also follows principles. Here are the key elements of a good prompt:

Table 1. Good prompt

| Element | Example |
|---|---|
| **Hardware environment** | FRDM-MCXA346 board, 180-MHz core, 256-kB RAM |
| **Functional requirements** | Implementing a 1-kB ring buffer for UART reception, thread-safe |
| **Implementation approach** | Using DMA, state machine pattern, FreeRTOS queue style |
| **Performance constraints** | Interrupt response time < 10 µs |
| **Interface specification** | API style follows MCUXpresso SDK, error handling via enum returns |
| **Code style** | Complies with MISRA C, uses Doxygen-format comments |

Most mainstream AI coding tools now offer prompt-optimization features to refine your prompts.

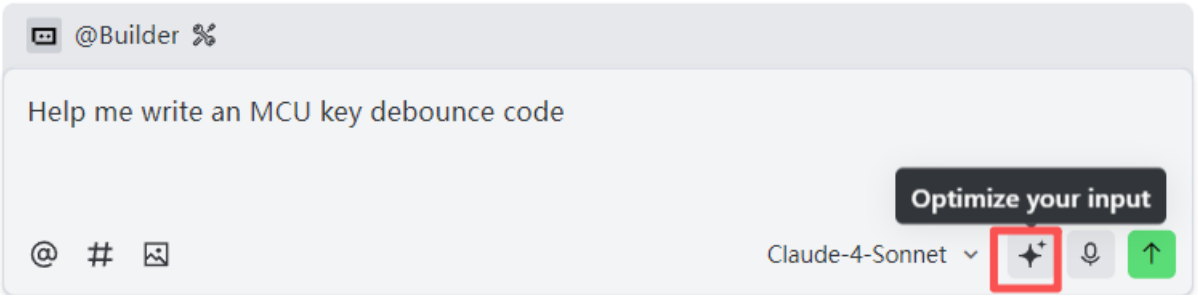


Figure 7. Prompt-optimization features

The context provides richer reference information, including:

- Chip data sheet key sections
- Schematic diagrams
- Pin-configuration tables
- Clock-tree configuration
- SDK API documentation
- Related source and header files

**Example**:

**Without context**: "Help me write SPI sensor reading code"

- AI: Provides a generic SPI read/write framework
- Developer: Compilation errors due to unknown SDK

**With rich context**: "I need SPI sensor reading implementation for FRDM-MCXA346 using Arduino interface SPI pins to drive the MPU6050 IMU sensor. Attached materials:

- [Upload 1] Sensor data sheet Chapter X (SPI timing and register definitions)
- [Upload 2] Existing project MPU6050 driver code (style reference)
- [Upload 3] MCUXpresso SDK fsl_lpspi.c, fsl_lpspi.h
- [Upload 4] MCUXpresso SDK polling_b2b_transfer example main.c
- Requirements: Reference the existing SDK SPI driver style, implement the MPU6050 sensor initialization and data reading"

Also, combine this with the "iterative validation" mindset mentioned earlier. Provide the context progressively rather than dumping all files at once:

1. Basic environment → "Using MCXA346 + MCUXpresso SDK" + SDK key headers
2. Related code → "Existing similar driver implementation" + current driver code
3. Specific requirements → "New functionality needed" + new device data sheet sections

AI excels at "learning by example." If you have similar code, provide it as a reference: "Our project's SPI driver is written like this (see attachment) and successfully drives XXXX. Now, I need an I²C driver. Follow the SPI driver's code style and interface design to implement it."

**Why does it work?**

AI operates as a "probability predictor" rather than a "search engine." It predicts the most likely next token based on context. More precise information increases the probability of correct predictions.

Imagine a fill-in-the-blank question: "In embedded systems, to prevent button bounce, _____ processing is usually required."

An engineer would fill in "debounce" or "debouncing" because we understand the context. AI works similarly, but it learns from massive amounts of code and documentation during training, learning "what code is most likely to appear in what context." Every character that AI generates depends on all preceding content (prompt + context + previous AI output). The more precise information you provide, the higher the probability that AI predicts the correct answer.

The role of prompts and context is equivalent to **narrowing the solution space**. Programming problems have countless solutions. For example, "implement a delay function" could be:

- Empty loop delay
- Timer-based delay
- System tick delay
- DMA + timer zero-CPU delay

When you only say "write a delay function", AI faces a huge "solution space" and can only give the most generic (often the least applicable) answer.

But when you say: "In a FreeRTOS environment, use vTaskDelay to implement a nonblocking delay function accurate to milliseconds, for periodic sensor data acquisition, requiring the lowest power consumption."

You have done three things:

1. **Narrowed the solution space**: Specified FreeRTOS API, ruling out other approaches
2. **Provided constraints**: Millisecond accuracy, nonblocking, low power - these are "filters"
3. **Given application context**: Periodic acquisition - AI considers the best related practices

AI faces a smaller "solution space" and the probability of predicting the correct answer increases greatly. The time invested in writing good prompts and preparing context can save multiples of debugging time.

## 4.4 Code snippets are more effective than text descriptions

AI typically understands code better than natural language due to programming language precision and lack of ambiguity.

**Example**:

- Text description: "Our project-error-handling style follows this pattern..." [500 words of description]
- Code snippet:

```
typedef enum {
    STATUS_OK = 0,
    STATUS_ERROR_PARAM,
    STATUS_ERROR_TIMEOUT,
    STATUS_ERROR_BUSY
```

```
} status_t;
status_t driver_init(driver_handle_t *handle) {
    if (handle == NULL) return STATUS_ERROR_PARAM;
    // ...
}
Please implement the new driver following this style."
```

A 5-line code snippet often provides more value than 500 words of description.

# 5 AI-suitable tasks

Understanding AI capability boundaries (knowing what it can and cannot do) maximizes the tool value.

## 5.1 Generic algorithm implementation

The algorithm implementation consists of the following principles:

- Standard communication protocols (Modbus, CAN parsing, JSON processing)
- Classic algorithms (CRC/checksum, filters, PID control)
- Data structures (ring buffers, linked lists, state machines)
- String processing (command parsing, format conversion)

**Why does it work?**

These principles represent "public knowledge" with extensive samples in AI-training data sets.

**Example**:

AI excels at implementing the CRC16-MODBUS:

- You: "Help me implement the CRC16-MODBUS checksum algorithm."
- AI: Immediately provides standard implementation, even with a lookup-table-optimized version.

AI struggles with the new MCU register configuration:

- You: "Help me configure the MCXA346's LPSPI1 using DMA channel 3."
- AI: Without precise context, it may give incorrect register addresses or nonexistent macro definitions.

**Rule of thumb**: If 100+ similar implementations exist on GitHub, AI performs well. If they are only in chip manuals, detailed context is required.

## 5.2 Code review and refactoring

AI's code-analysis capabilities often exceed generation capabilities because understanding provides rich context when compared to creation.

Your code:

```
void uart_send(uint8_t *data, uint16_t len) {
    for (int i = 0; i < len; i++) {
        while (!(UART->SR & UART_SR_TXE));  // ← AI will note: no timeout
 protection
        UART->DR = data[i];
    }
}
//
```

AI's suggestion:

```
status_t uart_send(const uint8_t *data, uint16_t len) {  // ← Add const
len == 0) return STATUS_ERROR_PARAM;  // ← Parameter check
    uint32_t timeout;
    for (uint16_t i = 0; i < len; i++) {  // ← Type match
        timeout = TIMEOUT_COUNT;
        while (!(UART->SR & UART_SR_TXE) && --timeout);  // ← Timeout protection
        if (timeout == 0) return STATUS_ERROR_TIMEOUT;
        UART->DR = data[i];
    }
    return STATUS_OK;
}
```

## 5.3 Documentation generation

AI excels at handling tedious tasks, such as documentation generation:

**Example 1: Automatic Doxygen comments**:

Your code (no comments):

```
status_t ring_buffer_write(ring_buffer_t *rb, const uint8_t *data, size_t len) {
    if (rbPARAM;
    // ... implementation ...
}
```

After AI adds comments:

```
/**
 * @brief Write data to ring buffer
 *
 * @param[in,out] rb    Ring buffer handle
 * @param[in]     data  Pointer to data to write
 * @param[in]     len   Data length in bytes
 *
 * @return status_t
 *   @retval STATUS_OK           Write successful
 *   @retval STATUS_ERROR_PARAM  Parameter error (null pointer)
 *   @retval STATUS_ERROR_FULL   Insufficient buffer space
 *
 * @note This function is not thread-safe; external locking required in multi-
threaded environments
 * @warning Returns error if write length exceeds available space; no partial
 writes
 */
status_t ring_buffer_write(ring_buffer_t *rb, const uint8_t *data, size_t len);
```

**Example 2: Generating module-level documentation**:

- You: "Based on this driver file, generate a README.md including: feature description, API list, and usage examples."
- AI: Generates structured documentation including initialization flow, typical call sequences, and notes.

Key insight: AI excels at common0knowledge tasks, but it needs human judgment for hardware-specific work.

- AI handles high-level code and standard algorithms well (protocols, CRC, and so on). but it struggles with low-level MCU drivers, especially the register configuration for new chips.

- AI excels at reviewing, optimizing, checking errors, and adding comments to existing code (which you feed as context). This is equivalent to providing highly relevant context material.
- AI is good at error-log analysis and compilation-error checking if you feed it the error code snippets and logs.
- AI excels at adding comments to existing code and generating documentation, saving significant time on mechanical repetitive labor.

# 6 Summary

Table 2 summarizes the key points to note when providing prompts and context:

**Table 2. Key points**

| Pitfall | Problem | Corresponding best practice | Correct approach |
|---|---|---|---|
| Assuming AI should know | Writing a timer interrupt | Emphasize prompts and context | On MCXA346, use CTIMER0 for a 1-ms timer interrupt for system timebase. |
| Requesting too much at once | Writing a complete Modbus protocol stack | Iterative validation | First implement basic UART communication, then add Modbus frame parsing, and finally add protocol logic. |
| Not providing existing code | Always having AI start from scratch | Emphasize prompts and context | Provide similar project code as a style reference |
| Not reviewing AI output | Assuming that working AI code is bug-free | - | Even functional code requires human review. Working code may contain obvious bugs or overengineering. |

The predictable trends in AI-assisted embedded programming are as follows:

- **Enhanced embedded understanding**: AI comprehends chip manuals, hardware schematics, and oscilloscope waveforms better.
- **Intelligent tool evolution**: Progress from code completion to requirement implementation, describe the desired functionality for the AI to plan, code, and test autonomously.
- **Reduced costs**: Current paid subscriptions may evolve into basic infrastructure-like search engines.
- **Specialized embedded AI models**: Train data sets, including extensive chip manuals, application notes, and hardware designs.
- **Multimodal interaction**: Draw timing diagrams or photograph circuit boards for the AI to generate driver code.

However, embedded tasks that require deep physical world understanding (hardware debugging, performance optimization, power optimization) remain primarily human-driven in the near term.

AI serves as a powerful tool for accelerating idea-to-code transformation. The true value lies in clearly understanding requirements and underlying problem principles. The ability to comprehend the problem essence remains the most valuable asset for engineers.

AI enhances development efficiency but cannot replace fundamental engineering judgment, hardware understanding, and system-level thinking that define professional embedded systems development.

# 7 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

# 8 Revision history

**Table 3. Revision history**

| Document ID | Release date | Description |
|---|---|---|
| AN14859 v.1.0 | 5 November 2025 | • Initial version |

# Legal information

## Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

## Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**HTML publications** — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** — NXP B.V. is not an operating company and it does not distribute or sell products.

## Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

# Contents