

Roy Zhuang

Abstract

This document was translated from a simplified Chinese source. (ZHCAGB3)

As AI technology brings profound transformation to the tech industry, particularly with the emergence of large models, it has greatly facilitated users in debugging and parsing code with AI tools. This article uses general-purpose AI large models to summarize a replicable debugging methodology that helps customers break free from debugging work in complex and cumbersome scenarios, enabling better debugging of Ti radar chips. It also presents an engineering-deployable AWR2944LC AI debugging solution, generates Skills and test scripts, and uses an AI Agent to perform EVM board connection, image loading, and dual-core control. Memory is read using both AI tools and CCS for comparative verification of correctness and feasibility, and test cases that more closely reflect real user scenarios are provided.

Table of Contents

1 Basic Principles of AI-Automated Debugging	2
2 Agent, Skill, and Prompt	4
3 Verifying the AI Agent	7
3.1 Generating the ccxml File and JS Script	7
3.2 Test Verification	9
4 Using the AI Agent to Assist Debugging	13
4.1 Simulating Stress Testing	13
4.2 Simulating Scenario Testing	15
5 Conclusion	17
6 References	17

List of Figures

Figure 1-1. ccxml file structure	2
Figure 2-1. Agent executing the command to read XDS110	4
Figure 2-2. Agent using SKILL.md	5
Figure 2-3. Prompt awr2944lc_debug_prompt.md	6
Figure 3-1. JS script generated by the Agent for loading an image	8
Figure 3-2. Point cloud results in the wave-transmitting state	9
Figure 3-3. Skill folder generated by the AI Agent	10
Figure 3-4. Test invocation workflow	10
Figure 3-5. Agent execution process	10
Figure 3-6. APIs actually invoked by the AI Agent	11
Figure 3-7. The first 16 words of 0xA5000000 read by the AI Agent	11
Figure 3-8. The first 16 words of 0xA5000000 read by the AI Agent	12
Figure 3-9. CCS reading 0xA5000000 Memory	12
Figure 4-1. Stress-testing AI Agent prompt and execution log	13
Figure 4-2. AI Agent simulated stress-testing log and results	14
Figure 4-3. AI Agent JS script execution log	14
Figure 4-4. Test case output results	15
Figure 4-5. AI Agent simulated scenario testing	16
Figure 4-6. Test log generated by the AI Agent	16
Figure 4-7. Logs of test results for the five test scenarios	16

Trademarks

All trademarks are the property of their respective owners.

1 Basic Principles of AI-Automated Debugging

The core of AI debugging of EVM boards lies not in natural language generation itself, but in the synergy between a configurable connection system and a programmable execution system. The former addresses whether the target hardware can be connected accurately, while the latter addresses whether the debugging process can be executed stably after connection. Together, they enable traditional board-level debugging, which relies on individual experience, to be transformed into a structured, reproducible engineering workflow. Therefore, the value of AI in this scenario is not to replace low-level debugging tools, but to perform workflow orchestration, state control, and result archiving within the constraints of the existing toolchain, thereby improving efficiency and consistency.

In TI device debugging, ccxml is the hardware debugging topology description file. It defines the JTAG debugger type (e.g., XDS110), the target device model (e.g., the AWR2944 family), the accessible cores (R5F, DSP, M4, etc.), the connection path, and session initialization parameters. In essence, ccxml provides a complete definition of connection object, connection method, and access boundary. The stability of a debugging session depends on whether the ccxml configuration matches the actual hardware topology. A mismatch may result in connection failures, unstable sessions, or abnormal core control. Therefore, a correctly configured ccxml file is a prerequisite for reliable automated debugging.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--
    通过 XDS110 连接 AWR2X44P 的 CCS 目标配置。
    本文件定义：
    1) 调试探针连接类型；
    2) 扫描挂接链路中的调试驱动；
    3) 目标器件平台定义。
-->
<configurations XML_version="1.2" id="configurations_0">
    <!-- 单个命名配置容器。 -->
    <configuration XML_version="1.2" id="configuration_0">
        <!-- 选择物理调试探针定义（XDS110）。 -->
        <instance XML_version="1.2" desc="Texas Instruments XDS110 USB Debug Probe" href="connections/TIXDS110_Connection.xml"
            id="Texas Instruments XDS110 USB Debug Probe" xml="TIXDS110_Connection.xml" xmlpath="connections"/>
        <!-- 将该配置绑定到一个具体的探针连接实例。 -->
        <connection XML_version="1.2" id="Texas Instruments XDS110 USB Debug Probe">
            <!-- 底层调试状态机后端驱动。 -->
            <instance XML_version="1.2" href="drivers/tixds510debugssm.xml" id="drivers" xml="tixds510debugssm.xml" xmlpath="drivers"/>
            <!-- CoreSight DAP 访问通道（ARM 调试路由必需）。 -->
            <instance XML_version="1.2" href="drivers/tixds510cs_dap.xml" id="drivers" xml="tixds510cs_dap.xml" xmlpath="drivers"/>
            <!-- Cortex-R 内核驱动（对应 mmWave 器件上的 MSS 类核心）。 -->
            <instance XML_version="1.2" href="drivers/tixds510cortexR.xml" id="drivers" xml="tixds510cortexR.xml" xmlpath="drivers"/>
            <!-- Cortex-M 内核驱动（若所选器件拓扑包含该核心）。 -->
            <instance XML_version="1.2" href="drivers/tixds510cortexM.xml" id="drivers" xml="tixds510cortexM.xml" xmlpath="drivers"/>
            <!-- ETB/CoreSight 跟踪能力支持。 -->
            <instance XML_version="1.2" href="drivers/tixds510etbcs.xml" id="drivers" xml="tixds510etbcs.xml" xmlpath="drivers"/>
            <!-- TI SoC JTAG 链中的 ICEPick 路由与控制。 -->
            <instance XML_version="1.2" href="drivers/tixds510icepick_m.xml" id="drivers" xml="tixds510icepick_m.xml" xmlpath="drivers"/>
            <!-- C66x DSP 内核驱动（DSS 类核心）。 -->
            <instance XML_version="1.2" href="drivers/tixds510c66xx.xml" id="drivers" xml="tixds510c66xx.xml" xmlpath="drivers"/>
            <!-- 本连接对应的目标平台/器件选择。 -->
```

Figure 1-1. ccxml file structure

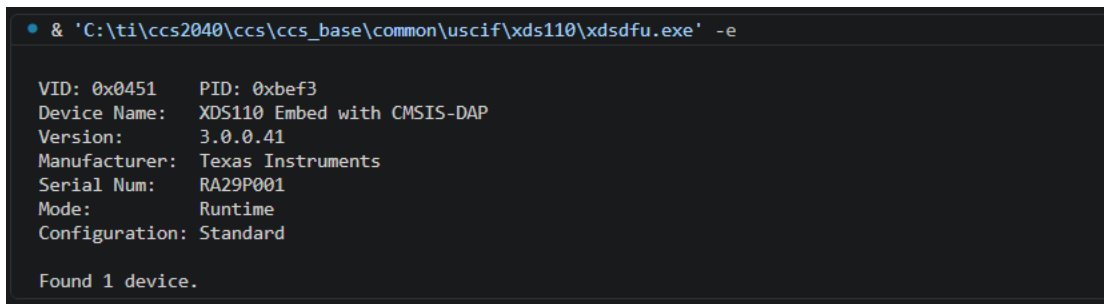
Automated execution is typically carried by JS scripts. Through the CCS Scripting interface, scripts invoke standard debugging primitives (such as connect, reset, loadProgram, run, halt, read, and log collection), converting manual steps into repeatable instruction sequences. Scripts do not replace the underlying debugging protocol; rather, they formalize debugging intent, giving the workflow engineering attributes of traceability, verifiability, and batch processing.

In short, ccxml defines where to connect and how to connect, JS scripts define what to do after connection, and AI automatically generates or modifies scripts and executes them according to user objectives, thereby replacing highly repetitive, low-value manual operations. A specific case is provided to verify the overall workflow. The AI tool used for testing in this article is the GPT-5.3-Codex Agent.

2 Agent, Skill, and Prompt

In automated debugging tasks, the key difficulty is executing stably in the correct order and closing the loop. JTAG identification, target connection, image loading, exception recovery, data reading, and result verification are essentially a continuous execution chain. To ensure the stability of this chain, Agent, Skill, and Prompt must be introduced simultaneously, corresponding respectively to the execution subject, execution rules, and task input.

The Agent is the executor. It not only provides recommendations but also completes steps in an environment where permissions allow, and continues to advance based on intermediate results. Taking 2944LC EVM board debugging as an example, the Agent can first check XDS110 visibility, then establish a session, load the image, run the program, collect output, and handle retries. Compared with a conversational mode that only provides operational suggestions, the Agent can form a closed loop of "execution completed and results returned". We need to use the Agent to help us implement the process of connecting to the XDS110, connecting to the chip core, and executing JS scripts. As shown in [Figure 2-1](#), the Agent determines the information of the XDS110 used on the EVM and the chip information by executing the command & 'C:\ti\ccs2040\ccs\ccs_base\common\uscif\xds110\xdsdfu.exe' -e, preparing for subsequent connection.



```
& 'C:\ti\ccs2040\ccs\ccs_base\common\uscif\xds110\xdsdfu.exe' -e

VID: 0x0451      PID: 0xbef3
Device Name:    XDS110 Embed with CMSIS-DAP
Version:        3.0.0.41
Manufacturer:   Texas Instruments
Serial Num:     RA29P001
Mode:           Runtime
Configuration:  Standard

Found 1 device.
```

Figure 2-1. Agent executing the command to read XDS110

A Skill provides engineering-level constraints on the execution procedure. It explicitly defines what to do at each stage, how to recover from failures, and how results should be presented. For example: first perform a JTAG visibility check; when encountering Error -260, first handle the occupying process^[1]; memory reads use the same expression convention as the CCS GUI; output is standardized to a fixed 32-bit hexadecimal format. The Skill reduces behavioral variability, improves result reviewability, and unifies output formats. Without Skill, the Agent may still complete the task, but its behavior is unstable and the results are difficult to review; with Skill, any task can proceed according to the prescribed workflow, and regardless of success or failure, a deterministic behavior will be executed.

```

---
name: awr2944lc-debug
description: 连接AWR2944LC、处理XDS110连接问题、加载MSS/DSS镜像、按CCS GUI一致口径读取内存（unsigned int* expression）。关键词：
AWR2944LC, XDS110, ccxml, loadProgram, memory read, CCS GUI, 32-bit hex
---
# AWR2944LC 调试 Skill
## 适用场景
- 需要通过 Agent 在真实硬件上连接 AWR2944LC。
- 需要加载 MSS/DSS 镜像并读取特定内存。
- 需要与 CCS GUI Memory Browser 的 32-bit 显示对齐。

## 强制规则
1. 优先保证探针可见：先跑 `xdsdfu -e`。
2. 若出现 `Error -260`，先检查并结束 `DSLite.exe`，再重试。
3. 读取结果若要和 CCS GUI 对齐，必须使用表达式读取：
   - `*((unsigned int*)0xADDR)`
4. `memory.read(addr,count)` 仅用于原始字节诊断，不作为 GUI 对齐结果。
5. 默认输出格式使用 **纯 32-bit Hex（8位、大写、无 `0x` 前缀）**，例如：`044A053A`。
6. 回传结果时必须带地址，格式：`0xADDR HEX32`。

## 标准流程
1) 连接并加载镜像（MSS + DSS）
2) 用户下发 cfg / 发波
3) 连接 MSS, halt
4) 逐地址 expr 读取 word
5) 以 `0xADDR XXXXXXXX` 输出（默认无 `0x` 值前缀）

## 防歧义口令（建议）
- 当用户明确要某一地址的值时，必须同时给出：
  - 地址（例如 `0xA5000000`）
  - word 数（例如 `1`）
  - 输出格式（`HEX32_8_NO_PREFIX`）
- 示例：
  - 请读取 0xA5000000, word=1, 仅输出 HEX32_8_NO_PREFIX。

## 参考命令
```powershell
$node='C:\ti\ccs2040\ccs\ccs_base\cloudagent\node.exe'
$launcher='C:\ti\ccs2040\ccs\scripting\launcher.mjs'
& $node $launcher 'C:\Users\<you>\workspace_ccstheia\mss_read_16words_expr_uint32.js'
```

## 失败恢复
- XDS110 不可见：重插 USB / 杀 `DSLite` / 再跑 `xdsdfu -e`
- C盘空间不足导致 ccxml cache 失败：清理 CCS 本地缓存目录
- 会话卡住：先 `halt` 再读；必要时重连会话

```

Figure 2-2. Agent using SKILL.md

Prompt is the input protocol for a single task. It defines the parameters of the current task, such as the connection object, whether to load MSS or DSS, the address to read, the number of words, the output format, the failure-retry sequence, and the final deliverables. A clear Prompt can significantly reduce ambiguity and lower the cost of repeated prompting. For example, when reading the memory of the AWR2944LC, the configuration parameters must be specified in `awr2944lc_debug_prompt.md`, as illustrated in [Figure 2-3](#). These parameters include the connection target, whether to load MSS or DSS, the address to read, the number of words, the output format, the failure-retry sequence, and the final deliverables. Such a Prompt directly reduces ambiguity and can also reduce the prompts given to the Agent each time the program is re-executed.

```
# AWR2944LC Agent Prompt 模板（可直接复制）
请连接 AWR2944LC 并执行以下任务：
1. 先检查 XDS110 是否可见（xdsdfu -e），若不可见，自动排查并恢复（含终止占用进程）。
2. 使用 `AWR2944P.ccxml` 连接目标。
3. 如我要求，先加载 MSS/DSS 镜像（`out_of_box_2x44LC_mss.xer5f` / `out_of_box_2x44LC_dss_cm4.xem4`）。
4. 读取内存时请使用 **CCS GUI 对齐口径**：
   - `*((unsigned int*)0xADDR)`
5. 输出格式固定为：
   - `0xAAAAAAAA VVVVVVVV`（默认纯8位HEX32，无 `0x` 前缀）

当前读取参数：
- 起始地址：`0xA5000000`
- word 数：`16`
- 输出格式：`HEX32_8_NO_PREFIX`

避免歧义（强烈建议一起给）：
- 指定地址：例如 `0xA5000000`
- 指定 word 数：例如 `1`
- 指定输出口径：`HEX32_8_NO_PREFIX`
示例（可直接复制）：
- `请halt后读取 0xA5000000，word=1，仅输出 HEX32_8_NO_PREFIX。`

- 期望形态：`0xA5000000 044A053A`
如果失败，请按以下顺序自动重试：
- 终止 `DSLite.exe`
- 重新枚举 XDS110
- 重新连接会话
- 再读取

最后给我：
- 实际执行日志摘要
- 读取结果
- 失败时的根因与下一步建议
```

Figure 2-3. Prompt awr2944lc_debug_prompt.md

3 Verifying the AI Agent

To verify the correctness of the Agent's execution, the Agent and CCS are used to compare the memory at the same address. If the two results are consistent, it can be proven that the key processes—connection, loading, halting, and reading—are correct and use the same conventions.

3.1 Generating the ccxml File and JS Script

The preparatory work consists of two parts: first, generating the ccxml corresponding to the AWR2944P family target; second, generating a JS script that covers the complete action chain (Connect Core, CPU Reset, Load Memory, Halt CPU, Read Memory, Save Memory).

The ccxml can be automatically generated by CCS according to the target device configuration, so it will not be elaborated here.

The JS script is not created out of thin air; rather, it is structurally composed and parametrically extended based on TI's official scripting patterns. Specifically, the Agent first studies the example scripts in the CCS scripting directory, extracts stable and reusable API invocation paradigms and exception-handling practices, and then organizes the actions into an executable state-machine workflow according to the user's objectives. On this basis, the script needs to be explicitly parameterized, including the ccxml path, image path, read address, read length, result save path, and timeout threshold, so that it can be reused across different projects, different images, and different test rounds without repeatedly modifying the main logic. A JS script that covers the complete debugging action chain. The complete action chain should include at least six steps—Connect Core, CPU Reset, Load Memory, Halt CPU, Read Memory, and Save Memory—organized in causal order rather than simply concatenating commands. [Figure 3-1](#) shows the JS script generated by the Agent, which can implement the simple function of Load Image. All interfaces use APIs from the example JS scripts, combined for specific scenarios.

```

const CCXML = "C:\\Users\\ workspace_ccstheia\\AWR2944P.ccxml"; // 目标连接配置 (ccxml)
const MSS_OUT = "C:\\ out_of_box_2x44LC_mss\\Debug\\out_of_box_2x44LC_mss.xer5f"; // MSS 镜像路径 (R5F)
const DSS_OUT = "C:\\ out_of_box_2x44LC_dss_cm4\\Debug\\out_of_box_2x44LC_dss_cm4.xem4"; // DSS 镜像路径 (CM4)
const MSS_BOOT_WAIT_MS = 3000; // MSS 启动后等待时间, 用于给 DSS 相关初始化留出时间
function openSession(ds, patterns, name) { // 按候选 pattern 依次尝试打开会话, 适配不同工程/版本的核命名差异
    for (const p of patterns) {
        try { return ds.openSession(p); } catch (e) {}
    }
    throw new Error(name + " session not found");
}
const ds = initScripting(); // 初始化 CCS 脚本调试引擎
ds.setScriptingTimeout(60000); // 设置默认超时, 避免连接/加载阶段超时过快
try {
    console.log("[1] configure ccxml");
    ds.configure(CCXML); // 加载目标连接配置, 这是后续 connect 的前提

    console.log("[2] connect MSS"); // 先连接 MSS (R5F), 作为主控核先完成加载与启动
    const mss = openSession(ds, [/Cortex_R5F_0/i, /Cortex_R5F/i, /Cortex_R5/i], "MSS");
    mss.target.connect();

    console.log("[3] reset + load MSS");

    try { mss.target.reset(); } catch (e) {} // 尝试复位 MSS, 保证加载前核心状态干净 (失败不阻断展示流程)
    mss.memory.loadProgram(MSS_OUT); // 加载 MSS 镜像到目标内存
    console.log("[4] run MSS");

    try { mss.target.run(); } catch (e) {} // 启动 MSS, 让其执行初始化流程
    sleep(MSS_BOOT_WAIT_MS); // 等待 MSS 启动并释放/准备 DSS 相关状态
    console.log("[5] connect DSS");

    const dss = openSession(ds, [/M4SS0_0/i, /Cortex_M4_0/i, /Cortex_M4/i, /CM4/i], "DSS"); // 再连接 DSS (CM4)

    dss.target.connect();
    console.log("[6] load DSS");

    dss.memory.loadProgram(DSS_OUT); // 加载 DSS 镜像
    console.log("[7] run MSS + DSS");

    try { mss.target.run(); } catch (e) {} // 最终确保两个核心都处于运行状态
    try { dss.target.run(); } catch (e) {}

    console.log("[OK] load complete, both cores running");
} catch (e) {

    console.log("[ERROR] " + e); // 统一错误出口, 便于外层工具识别失败状态
    process.exitCode = 1;
} finally {
    try { ds.shutdown(); } catch (e) {} // 释放调试会话资源, 避免占用探针
}

```

Figure 3-1. JS script generated by the Agent for loading an image

As demonstrated, the AI Agent can learn how these APIs work by referring to the JS examples provided by CCS, then combine them as needed to meet user requirements and automate the repetitive task of writing and rewriting scripts.

3.2 Test Verification

The purpose of this test is to verify the reliability of the AI Agent and whether the overall workflow can be fully connected end to end. The Memory address read is 0xA500 0000, which stores ADC data, ensuring the randomness of the test data.

Test method:

1. Use instructions to have the AI Agent automatically connect to the AWR2944LC and perform reset, image loading, program halt, Memory read, and save.
2. Disconnect the AI Agent, keep the AWR2944LC EVM board state unchanged, and use CCS to connect to the AWR2944LC and read the memory at the same address.
3. By comparing whether the Memory obtained by the two methods is identical, it can be determined whether the overall workflow executed by the AI Agent is correct.

Hardware preparation: The AWR2944LC EVM board has already been flashed with the SDK default demo: awr2x44LC_mmw_demoDDM.appimage. After power-on, load the default cfg file to transmit waves, and it can be seen from the mmWave Demo Visualizer that the EVM board is working properly.

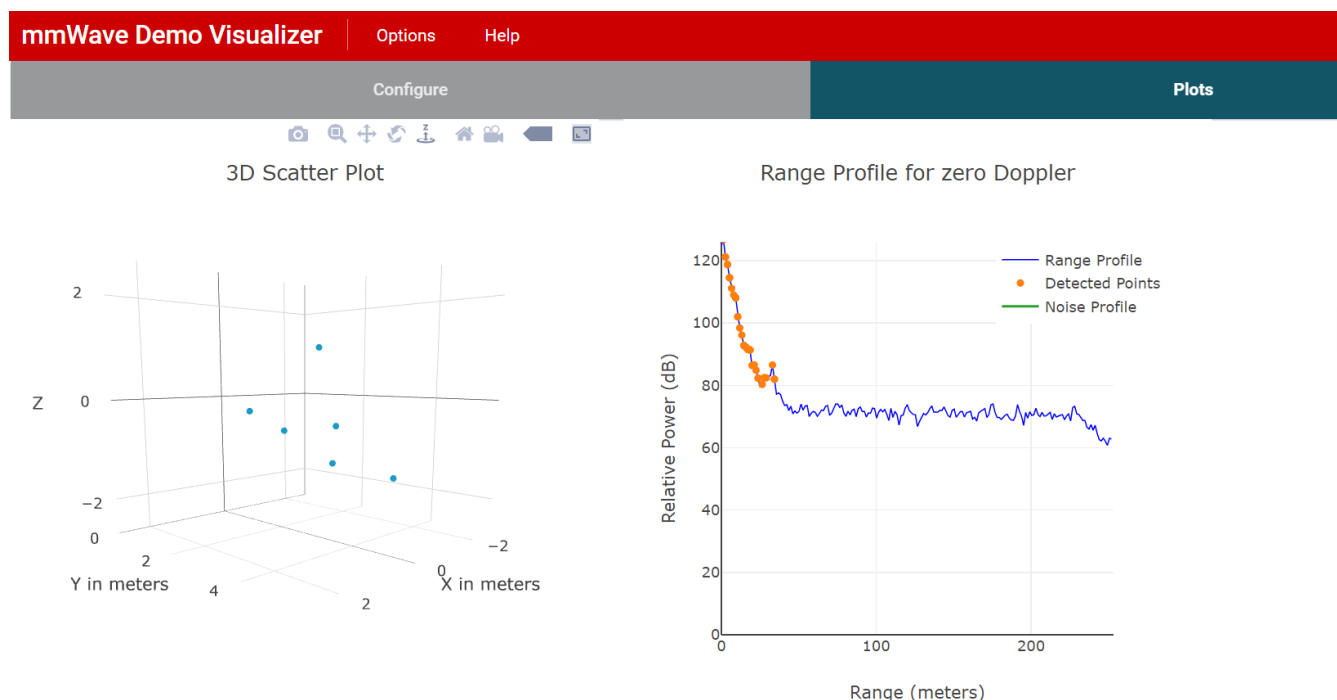


Figure 3-2. Point cloud results in the wave-transmitting state

At this point, use the AI Agent to input a prompt, and based on the AWR2944LC ccxml and C:\ti\ccs2040\ccs\scripting as references, generate a Skill for connecting to the AWR2944LC EVM and reading the first 16 words of Memory at 0xA500 0000. The generated Skill is as follows:

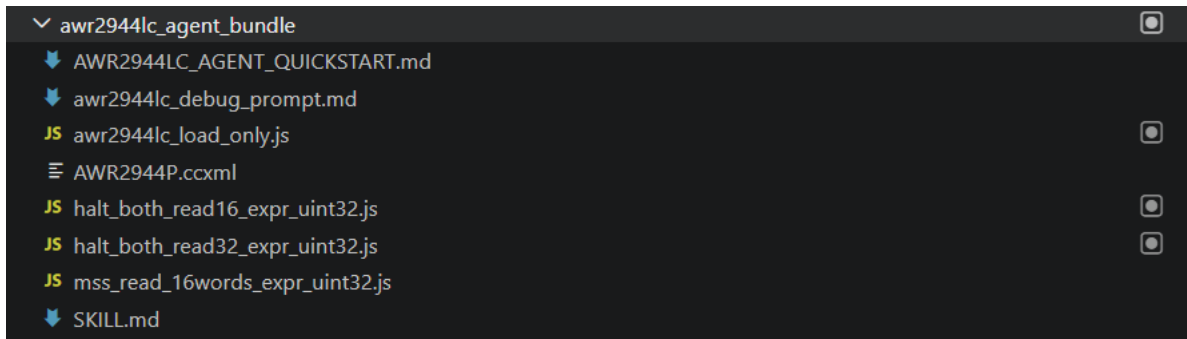


Figure 3-3. Skill folder generated by the AI Agent

Including:

1. Skill.md is used to specify the order in which the Agent executes tasks
2. Prompt.md is used to specify the default input prompt
3. JS script, the script generated by the Agent for executing actual tasks
4. AWR2944LC.ccxml, the configuration file for connecting to the XDS110
5. Quick Guide, the manual generated by AI on how to use the Skill

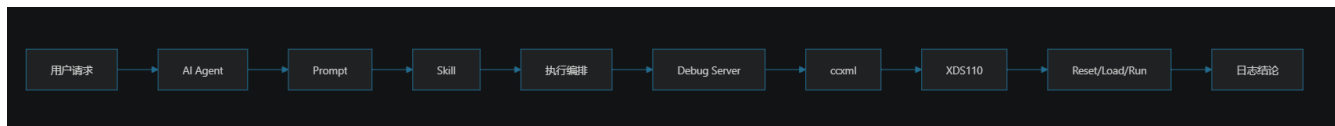


Figure 3-4. Test invocation workflow

Using this Skill, the AI Agent can quickly use the CCXML configuration file to connect to the AWR2944EVM and read the specified Memory:



Figure 3-5. Agent execution process

As shown in Figure 3-4, the upper right corner shows the generated Skill. Based on this Skill, the Agent reads the previously generated JS script and adds the Memory-reading portion to the JS. It can be seen that the Agent

executes the JS script, and according to the contents of the JS, the ccxml has already been configured into it, connecting to the MSS and DSS of the AWR2944LC through the XDS110 and halting them.

```

使用的脚本是 halt_both_read16_expr_uint32.js。

脚本内“连接 / halt / 读取”对应执行调用如下：

// 连接
ds.configure(CCXML);
const mss = openMss(ds);
const dss = openDss(ds);
mss.target.connect();
dss.target.connect();

// halt
mss.target.halt();
dss.target.halt();

// 读取 0xA5000000 起始 16 个 word (按 unsigned int* 口径)
const expr = "*((unsigned int*)0xA5000000)"; // 实际循环到 0xA500003C
const val = mss.expressions.evaluate(expr);

本次读取结果 (0xA5000000 起 16 words) :

```

Figure 3-6. APIs actually invoked by the AI Agent

The first 16 words of 0xA500 0000 read are as follows:

```

本次读取结果 (0xA5000000 起 16 words) :

0xA5000000  FF92FFF2
0xA5000004  FCB1FF47
0xA5000008  FF95FFF6
0xA500000C  FC50FF5B
0xA5000010  FFBCEFFD
0xA5000014  FC50FF72
0xA5000018  FFB3FFF4
0xA500001C  FC4EFF58
0xA5000020  FF7EFFF2
0xA5000024  FBC2FF34
0xA5000028  FFC1FFDD
0xA500002C  FC11FF46
0xA5000030  FF9CFFBF
0xA5000034  FBEDFF33
0xA5000038  FFACFFB3
0xA500003C  FBC5FF06

```

Figure 3-7. The first 16 words of 0xA5000000 read by the AI Agent

Read the first 32 words at the same address again to verify repeatability:

已读取 0xA5000000 起始前 32 个 word，结果如下：

```

0xA5000000 FF92FFF2
0xA5000004 FCB1FF47
0xA5000008 FF95FFF6
0xA500000C FC50FF5B
0xA5000010 FFBCFFDF
0xA5000014 FC50FF72
0xA5000018 FFB3FFF4
0xA500001C FC4EFF58
0xA5000020 FF7EFD2
0xA5000024 FBC2FF34
0xA5000028 FFC1FFDD
0xA500002C FC11FF46
0xA5000030 FF9CFFBF
0xA5000034 FBEDFF33
0xA5000038 FFACFFB3
0xA500003C FBC5FF06
0xA5000040 FFBAFF9B
0xA5000044 FC04FF30
0xA5000048 FFCCFF95
0xA500004C FC18FEED
0xA5000050 FFE0FF7F
0xA5000054 FC28FEC0
0xA5000058 FFDEFF61
0xA500005C FC0DFAF
0xA5000060 FFCDF48
0xA5000064 FC35FE90
0xA5000068 FFD0FF18
0xA500006C FC1DFA61
0xA5000070 FFE2FF17
0xA5000074 FBFEFE31

```

Figure 3-8. The first 16 words of 0xA5000000 read by the AI Agent

CCS: After using the AI Agent to capture the specified Memory, keep the board state unchanged and directly use CCS to connect to the AWR2944LC EVM, then locate the Memory at the corresponding address. The results are as follows:

```

0xA5000000 FF92FFF2 FCB1FF47 FF95FFF6 FC50FF5B FFBCFFDF FC50FF72 FFB3FFF4 FC4EFF58 FF7EFD2 FBC2FF34
0xA5000028 FFC1FFDD FC11FF46 FF9CFFBF FBEDFF33 FFACFFB3 FBC5FF06 FFBAFF9B FC04FF30 FFCCFF95 FC18FEED
0xA5000050 FFE0FF7F FC28FEC0 FFDEFF61 FC0DFAF FFCDF48 FC35FE90 FFD0FF18 FC1DFA61 FFE2FF17 FBFEFE31
0xA5000078 FFE5FF01 FC18FE2B FFD0FF07 FC3CFE0B FFE3FEF8 FC92FDE3 FFD2FED4 FC6DFDDE FFDFAFE8 FC65FDAC
0xA50000A0 FFDADFED8 FC5DFDAE FFEDFEBB FC66FD58 FFF8FED7 FCA9FD51 0000FED1 FCACFD4F 0003FEA5 FCA2FD25
0xA50000C8 000BFE8E FC79FD07 002BFE88 FCCDFD29 0036FE5E FCD0FD08 FFE8FE64 FCC2FCEF FFFBFE76 FCD8FD26
0xA50000F0 0033FE55 FD1FFD54 FFF5FE79 FD7FFD8C FFF6FE5E FD2EFD85 FFD8FE43 FCE0FD5A FFE6FE4C FD76FDB3

```

Figure 3-9. CCS reading 0xA5000000 Memory

The comparison results are consistent, indicating that the Agent workflow executes correctly and that the read convention is consistent with CCS. After repeated reads with an increased number of words, the results remain consistent, further verifying repeatability and stability.

4 Using the AI Agent to Assist Debugging

The previous chapters introduced the working principles of the AI Agent and verified the feasibility of using the AI Agent to debug the AWR2944LC. It can be seen that the AI Agent is better suited to tedious, complex, and highly repetitive work. In radar device debugging, there are a large number of highly repetitive, long-cycle tasks, such as repeatedly flashing and running different images, stress testing, reproducing intermittent faults, restarting to capture logs, and capturing memory. Such tasks are suitable for the Agent to take over, which can significantly reduce the burden of manual operations and improve statistical efficiency.

This article uses the default DPU test programs in the SDK as cases to verify that the AI Agent can perform repetitive testing work:

1. Repeatedly execute test case 20 times to simulate stress testing
2. Execute all DPU test cases to simulate scenario testing

4.1 Simulating Stress Testing

The program `dpu\rangecfarprocDDMA\test\awr2x44P_rangecfarprochwa_DDMA_m4.xem4` in the SDK is used as the test case. Reset, load image, run image, and save log are treated as one cycle, repeated 20 times to simulate a stress-testing scenario. As shown in [Figure 4-1](#), a simple prompt allows the AI Agent to write a JS script that meets the requirements and perform the test, with the test results statistically summarized and saved.

基于此Skill, 连接到板子后reset, 导入awr2x44P_rangecfarprochwa_DDMA_m4.xem4此镜像, 保存log, log中显示--- Test Completed ---后reset重新导入镜像, 进行下一次循环, 将20次执行的结果保存起来。

Optimized tool selection

我会先复用现有调试脚本能力, 新增一个“检测到 --- Test Completed --- 就结束本轮并reset”的执行脚本, 再写一个20次循环的批处理入口并直接帮你跑起来, 把每轮日志和汇总结果落盘。

Read `awr2x44p_m4_load_run_capture_cio.js`

下一步我会新增两个文件: 一个JS负责“运行并监测 CIO 关键字后立即结束+reset”, 一个PS1负责串行跑20轮并汇总通过率, 结果会保存在 `rangecfarprocDDMA/test/testdata`。

Created 2 files

我现在开始执行20轮实测: 每轮连接后 reset、导入 `awr2x44P_rangecfarprochwa_DDMA_m4.xem4`、检测 --- Test Completed ---、再reset进入下一轮, 并把单轮日志与总表落盘。

已运行 `$root='C:\ti\mmwave_mcuplus_sdk_04_07_01_03\mmwave...'`

```

• $root='C:\ti\mmwave_mcuplus_sdk_04_07_01_03\mmwave_mcuplus_sdk_04_07_01_03'; Set-Location $root; & 'C:\ti\mmwave_mcuplus_sdk_04_07_01_03\mmwave_mcuplus_sdk_04_07_01_03\ti\datapath\dpu\rangecfarprocDDMA\test\run_m4_rangecfar_20_loops.ps1'

[8] reset M4 after cycle
[8.1] reset ok
===== Loop 11 / 20 =====
[1] configure ccxml: C:\ti\mmwave_mcuplus_sdk_04_07_01_03\mmwave_mcuplus_sdk_04_07_01_03\ti\datapath\dpu\awr2944lc_agent_bundle\AWR2944P.ccxml
[2] open/connect M4
[3] reset M4 before load
[3.1] reset ok
[4] load image: C:\ti\mmwave_mcuplus_sdk_04_07_01_03\mmwave_mcuplus_sdk_04_07_01_03\ti\datapath\dpu\rangecfarprocDDMA\test\awr2x44P_rangecfarprochwa_DDMA_m4.xem
  
```

20轮任务已经跑完, 我现在读取运行结果文件并核对 `summary.csv` 与 `final_result.txt`, 然后给你最终统计和日志位置。

Figure 4-1. Stress-testing AI Agent prompt and execution log

It can be seen that the AI Agent can quickly understand and execute simple, repetitive work. [Figure 4-2](#) shows the test results; the AI Agent executes such tasks well and summarizes the results. In practical applications, when encountering similar problems—such as rare hang issues that are difficult to reproduce or long-duration

stress testing—this method can be used, greatly saving users the repetitive work of rewriting JS scripts and summarizing results.

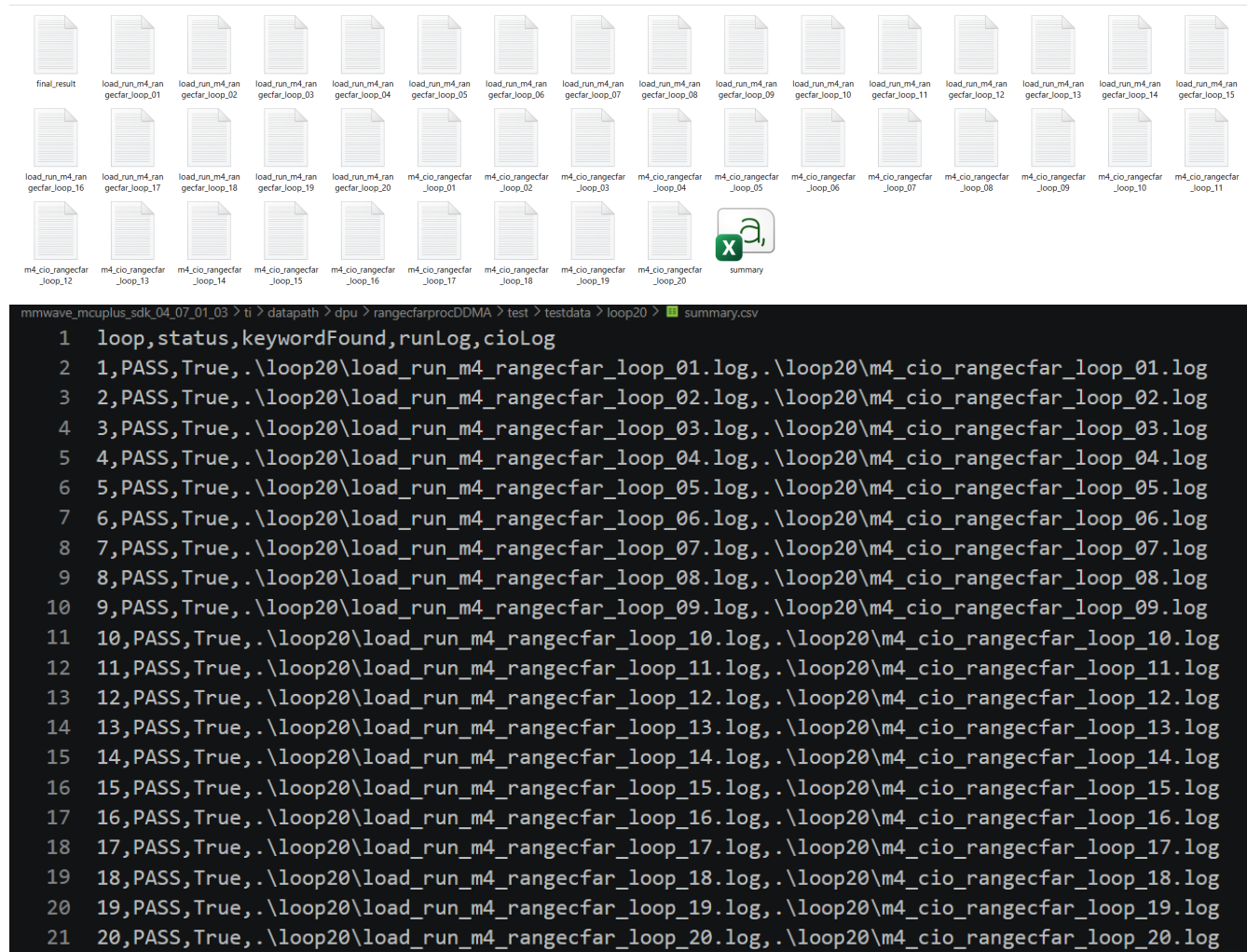


Figure 4-2. AI Agent simulated stress-testing log and results

The logs fall into two categories. One is the log generated during the AI Agent's execution of the JS script, used to monitor hardware status.

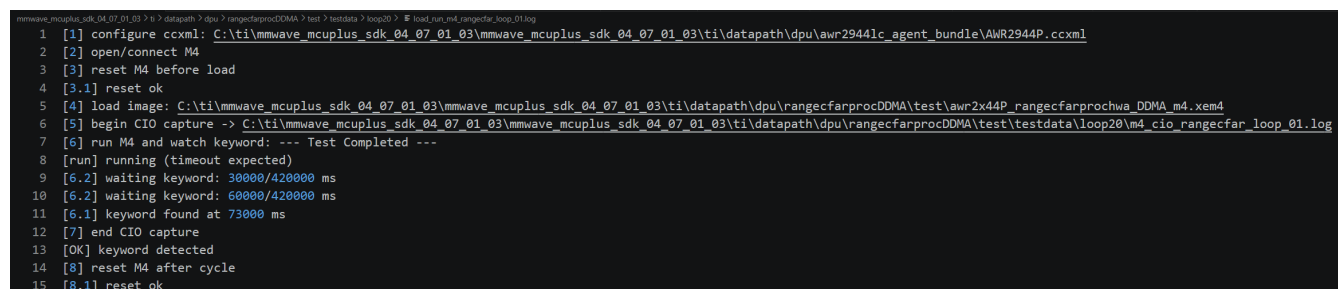


Figure 4-3. AI Agent JS script execution log

The other is the log of the test results when the actual test case is executed, as shown in Figure 4-4:

```
mmwave_mcuplus_sdk_04_07_01_03 > ti > datapath > dpu > rangeconfprocDDMA > test > testdata > loop20 > m4_cio_rangeconf_loop_01.log
1 ..... Initialization .....
2 #####
3
4 Test #0 start
5
6 -----
7 numRangeBins: 96
8 numDopplerBins: 96
9 cfarThreshold(dB): 3827
10 averageMode: 3
11 winLen: 16
12 guardLen: 0
13 -----
14 NumObjOut = 37, NumObjOutIdeal = 37
15
135
136 8 Tests finished!
137
138 Test Time Cycles: 22221
139 All Tests finished!
140
141 All tests have passed!!
142 .....TEST PASSES.....
143 --- Test Completed ---
144
```

Figure 4-4. Test case output results

4.2 Simulating Scenario Testing

In addition to repeated testing of a single scenario, the Agent is also suitable for single-pass verification across multiple scenarios. The following five default test cases under the SDK DPU are used for testing:

1. awr2x44P_aoaprochwa_mss.xer5f running on R5F
2. 2x44P_dopplerprochwa_mss.xer5f running on R5F
3. awr2x44P_rangeprochwa_mss.xer5f running on R5F
4. awr2x44P_rangeconfprocDDMA_m4.xem4 running on M4
5. awr2x44P_rangeprochwa_mss.xer5f running on R5F

Similar to Section 4.1, a simple prompt describing the task is sufficient to have the AI Agent execute it:

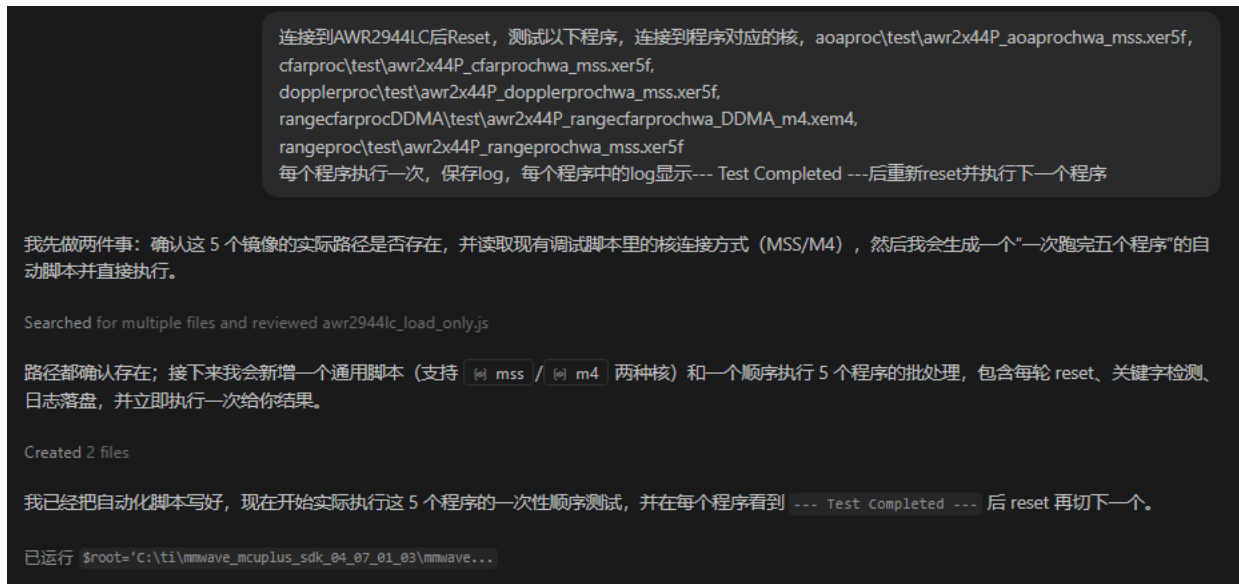


Figure 4-5. AI Agent simulated scenario testing

The generated results are as follows:

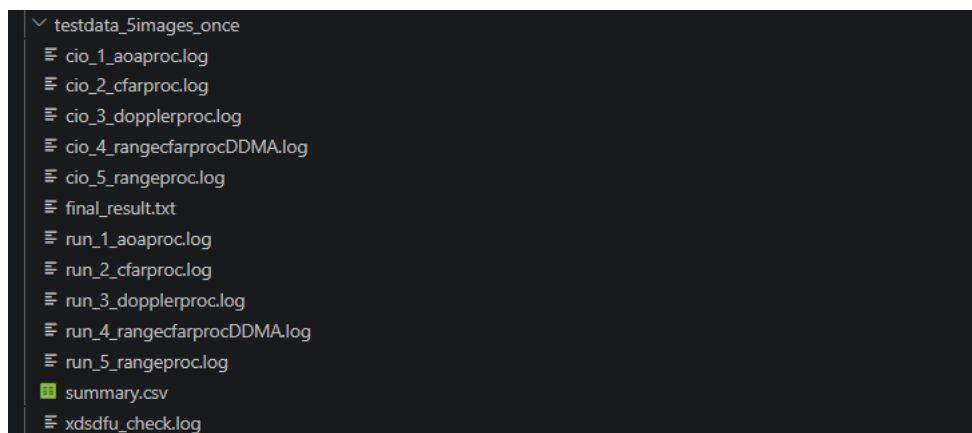


Figure 4-6. Test log generated by the AI Agent

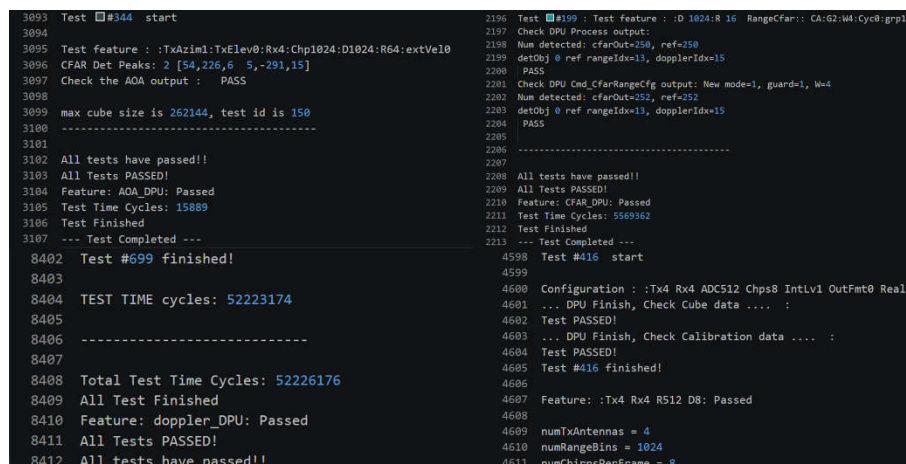


Figure 4-7. Logs of test results for the five test scenarios

From the results in Figure 4-7 above, it can also be seen that, in addition to repeated testing of a single scenario, the Agent is also suitable for single-pass verification across multiple scenarios. After batch execution of multiple default test cases, the Agent can automatically complete scheduling, log collection, and result archiving. The results show that the Agent also has good stability and usability in cross-scenario testing, facilitating the rapid implementation of complex verification tasks.

5 Conclusion

This article demonstrates the engineering feasibility of an AI Agent in AWR2944 series EVM debugging, serving as a starting point for further exploration. The key is not to replace underlying tools, but rather to use ccxml to provide a deterministic connection prerequisite, JS scripts to implement a stable execution chain, and Skill and Prompt to ensure consistent rules and clear input. Through alignment verification with CCS results, the Agent demonstrates accuracy and repeatability in key steps such as memory reading; in stress and multi-scenario testing, the Agent further exhibits advantages in automation, standardization, and traceability. For debugging tasks that are highly repetitive, complex in workflow, and require long-term operation, this solution has clear practical value.

6 References

1. [Debugging JTAG](#)
2. [Getting Started with AWR2944 Online Debugging](#)
3. [mmWave Demo Visualizer](#)
4. [AWR2944LC Datasheet](#)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#), [TI's General Quality Guidelines](#), or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025