A Citable Inventory of Platforms, Operating Systems, and Compiler Toolchains

Document Number:	P4182R1
Date:	2026-05-01
Intent:	Inform
Audience:	SG14, EWG, LEWG

Reply-to:	Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

Every mailing repeats the same deployment background. This paper puts one citeable inventory in the record.

It consolidates platform categories, operating-system and runtime context, and representative compiler families so authors can cite a single WG21 paper instead of rehearsing the same background in every proposal. The platform survey extends the capability table in P4127R0 [1] with additional columns aimed at SG14 readers. The compiler survey adds matching rows for toolchain families. The document is a **working reference**. Later revisions may extend the schema and add rows. Proposals with sources may reach the author through the addresses in the front matter.

Revision History

R1: May 2026 (pre-Brno mailing)

- Formatting corrections.

R0: April 2026 (post-Croydon mailing)

- Initial version.

---

1. Disclosure

The author is a contributor to the Network Endeavour (P4100R0 [2]) and maintains tooling in that programme. This paper is reference material. It requests no poll and no feature adoption. It exists so other authors have a stable citation for deployment and toolchain facts.

---

2. Motivation

Committee papers that depend on deployment facts - hosted versus freestanding libraries, `thread_local` behaviour, allocator and exception defaults, or which toolchain generations matter for a target - currently ask each author to rehearse the same background. That repetition wastes reader time and invites inconsistent summaries. It pulls focus from the technical claim under discussion.

This paper consolidates a single, citeable inventory of **platform categories**, the **operating-system and runtime context** those categories imply, and **compiler and toolchain families** that appear in SG14-relevant work. It is offered as a **courtesy to SG14** and to **any author** who needs a stable reference for those facts. Cite this paper when you need that background without writing it out again. You are not asked to treat it as normative beyond what the cited evidence supports.

The inventory is a **living document**. Later revisions may add schema columns, platform rows, and compiler rows as the record improves. Propose additions or corrections by contacting the author at the address in the front matter. Include sources the

editors can verify.

The platform half of the inventory extends the capability survey used in P4127R0 [1] Section 9.1 ("Platforms Without Thread-Local Storage") for a different question - deployment and toolchain context for authors, not the coroutine frame allocator timing argument itself. The compiler half fills the second table on the same footing: reference material, maintained over time.

---

3. Platforms

3.1 Platform schema

The schema below states baseline capabilities from P4127R0 [1] Section 9.1, plus additional columns for SG14 readers. It is presented as two tables sharing the **Category** index so each table has room to read clearly.

Table A - Language and library support. What C++ facilities apply on each platform.

Category	Coro	TLS	PMR	Heap	Hosted
Desktop (Linux, Windows, macOS)	Yes	Yes	Yes	Yes	Full
Mobile (iOS, Android)	Yes	Yes	Yes	Yes	Full
Game consoles (Xbox, PS5)	Yes	Yes	Yes	Yes	Full
Full RTOS (QNX, Zephyr, VxWorks)	Yes	Yes	Hosted PMR	Yes	Partial to full
Lightweight RTOS (FreeRTOS, Pico)	Partial	No	No	Yes	Freestanding
Bare metal (Cortex-M, RISC-V)	Partial	No	No	Rare	Freestanding
GPU device code (CUDA, SYCL)	No	No	No	No	N/A

Legend for Table A:

- **Category** - Row label from the Section 9.1 survey.
- **Coro** - Whether C++20 coroutines apply in practice for that category.
- **TLS** - Whether `thread_local` is available.
- **PMR** - Whether the hosted `<memory_resource>` facility applies where relevant [3].
- **Heap** - Whether a general-purpose heap worth customising exists for that category.
- **Hosted** - Usual standard-library profile (full, partial to full, freestanding, or N/A).

Table B - Runtime context. How code actually runs on each platform.

Category	Exc	Alloc	RT	TLSctx
Desktop (Linux, Windows, macOS)	Typically on	Heap + PMR	Best effort	Static TLS
Mobile (iOS, Android)	Typically on	Heap + PMR	Best effort	Static TLS
Game consoles (Xbox, PS5)	Often off	Arenas common	Best effort	Static or vendor
Full RTOS (QNX, Zephyr, VxWorks)	Varies	Pools common	Soft to hard RT	Static TLS
Lightweight RTOS (FreeRTOS, Pico)	Typically off	Static pools	Soft RT	N/A
Bare metal (Cortex-M, RISC-V)	Typically off	Static only	Hard or none	N/A
GPU device code (CUDA, SYCL)	N/A	Device heaps differ	SIMT	N/A

Legend for Table B:

- **Category** - Row label from the Section 9.1 survey.
- **Exc** - Exception default in typical shipping builds. Labels are **Typically on**, **Often off**, **Varies**, **Typically off**, or **N/A** when the column does not apply.
- **Alloc** - Dominant hot-path allocation style.
- **RT** - Scheduling class (best effort, soft or hard real-time, SIMT, or none).
- **TLSctx** - How thread-local storage is accessed when TLS exists (static TLS, vendor-specific, or N/A).

3.2 Desktop (Linux, Windows, macOS)

Executive summary

Desktop operating systems deliver full hosted implementations, C++20 coroutines in production toolchains, `thread_local`, PMR, and heaps that teams routinely customise. SG14 concerns (latency, allocation control) show up as `-fno-exceptions` or custom allocators in games and trading stacks, not as absence of the machinery.

Schema

Field	Value
Platform category	Desktop (Linux, Windows, macOS)
C++20 coroutines (in practice)	Yes
`thread_local` available	Yes
Hosted `<memory_resource>`	Yes
General-purpose heap worth customising	Yes
Hosted standard-library profile	Full
Exceptions in typical shipping builds	Typically on. Most shipping desktop binaries leave exceptions enabled unless a product or studio policy turns them off
Hot-path allocation style	Heap plus PMR
Scheduling class	Best effort. General-purpose OS scheduling (not a hard real-time kernel guarantee)
Thread-local storage access pattern	Static TLS. Thread-local objects live in the loader-managed TLS block for the process or main modules. On Windows, DLLs can still pay an extra indirection as discussed in P4127R0 [1] Section 9.2

P4127R0 [1] Section 9.1 groups this row with the same top-tier capability set as other full-hosted platforms in that survey.

3.3 Mobile (iOS, Android)

Executive summary

Mobile stacks are full hosted environments on ARM64 for mainstream C++ toolchains. The same capability row as desktop applies for the features this inventory tracks.

Schema

Field	Value
Platform category	Mobile (iOS, Android)
C++20 coroutines (in practice)	Yes
`thread_local` available	Yes
Hosted `<memory_resource>`	Yes
General-purpose heap worth customising	Yes
Hosted standard-library profile	Full
Exceptions in typical shipping builds	Typically on
Hot-path allocation style	Heap plus PMR
Scheduling class	Best effort
Thread-local storage access pattern	Static TLS

P4127R0 [1] Section 9.1 lists this row alongside desktop. Store rules and battery targets change optimisation pressure, but hosted builds keep the core facilities in the table.

3.4 Game consoles (Xbox, PS5)

Executive summary

Vendor SDKs supply hosted C++ with coroutine support and TLS for titles that opt in. Many shipping games disable exceptions and rely on arena-style allocation. The inventory row still records Yes for the core language and library features in the survey.

Schema

Field	Value
Platform category	Game consoles (Xbox, PS5)
C++20 coroutines (in practice)	Yes
`thread_local` available	Yes
Hosted `<memory_resource>`	Yes
General-purpose heap worth customising	Yes
Hosted standard-library profile	Full
Exceptions in typical shipping builds	Often off. Many titles compile with exceptions disabled for size and determinism even though the toolchain supports them
Hot-path allocation style	Arenas common. Frame and transient data served from arena or pool allocators rather than unbounded heap traffic on hot paths
Scheduling class	Best effort
Thread-local storage access pattern	Static or vendor. Access path can be the usual static TLS block or a vendor-specific runtime detail not visible in public documentation

P4127R0 [1] Section 9.1 includes this row in the top group. Treat vendor guidance as the authority for a specific title target.

3.5 Full RTOS (QNX, Zephyr, VxWorks)

Executive summary

These kernels support threaded C++ with TLS in configurations that ship coroutines and a hosted or partially hosted standard library. Allocation on hot paths is often pooled.

Schema

Field	Value
Platform category	Full RTOS (QNX, Zephyr, VxWorks)
C++20 coroutines (in practice)	Yes
`thread_local` available	Yes
Hosted `<memory_resource>`	Hosted PMR (platform-supplied). May appear as part of the platform-provided hosted subset, not necessarily a full desktop-class library layout. P4127R0 [1] Section 9.1 records the PMR column as **Hosted** for this row
General-purpose heap worth customising	Yes
Hosted standard-library profile	Partial to full. From slim RTOS images through near-desktop library surface
Exceptions in typical shipping builds	Varies. Automotive and industrial certification: some builds forbid exceptions entirely, others allow them in non-safety paths
Hot-path allocation style	Pools common
Scheduling class	Soft to hard real-time. Spans best-effort timing guarantees through kernels with bounded response times up to formally verified schedules
Thread-local storage access pattern	Static TLS

3.6 Lightweight RTOS (FreeRTOS, Pico)

Executive summary

Many microcontroller-focused RTOS images lack `thread_local` and lack hosted PMR even when coroutine libraries exist. Allocation is explicit and bounded.

Schema

Field	Value
Platform category	Lightweight RTOS (FreeRTOS, Pico)
C++20 coroutines (in practice)	Partial
`thread_local` available	No
Hosted `<memory_resource>`	No
General-purpose heap worth customising	Yes
Hosted standard-library profile	Freestanding
Exceptions in typical shipping builds	Typically off. Many embedded build configurations use `-fno-exceptions` or equivalent
Hot-path allocation style	Static pools. Fixed blocks and explicit allocators instead of a polymorphic PMR surface
Scheduling class	Soft real-time. Typical RTOS latency goals without claiming a single certification level
Thread-local storage access pattern	Not applicable

P4127R0 [1] Section 9.1 gives **Partial** for coroutines and **No** for `thread_local` and PMR. The Raspberry Pi Pico (RP2040) example in that section notes community coroutine support, incomplete `thread_local` in the SDK at the time of writing, and newlib in freestanding mode without `<memory_resource>` [1]. Pigweed provides C++20 coroutines with an allocator passed per coroutine (`CoroContext`) because PMR is not available in those environments [4].

3.7 Bare metal (Cortex-M, RISC-V)

Executive summary

Firmware without an OS may still use a constrained heap. Coroutines are partial and TLS is absent in the survey. SG14 workloads here assume deterministic, static allocation patterns.

Schema

Field	Value
Platform category	Bare metal (Cortex-M, RISC-V)
C++20 coroutines (in practice)	Partial
`thread_local` available	No
Hosted `<memory_resource>`	No
General-purpose heap worth customising	Rare. Many firmware images use only static storage and stack. Any heap is often a small custom arena
Hosted standard-library profile	Freestanding
Exceptions in typical shipping builds	Typically off
Hot-path allocation style	Static only. Dominantly static and stack storage on hot paths
Scheduling class	Hard real-time or none. Strict cycle-bound firmware is one case; bring-up or lab setups where no RTOS schedule applies are the other
Thread-local storage access pattern	Not applicable

P4127R0 [1] Section 9.1 marks heap as **Rare** for this row. The intersection of hosted PMR and absence of TLS remains empty in that paper's survey.

3.8 GPU device code (CUDA, SYCL)

Executive summary

Device compilation targets are not general hosted C++ environments. Coroutines, `thread_local`, and PMR as used in this inventory do not apply to SIMT device code in the same sense as host code.

Schema

Field	Value
Platform category	GPU device code (CUDA, SYCL)
C++20 coroutines (in practice)	No
`thread_local` available	No
Hosted `<memory_resource>`	No
General-purpose heap worth customising	No
Hosted standard-library profile	Not applicable
Exceptions in typical shipping builds	Not applicable
Hot-path allocation style	Device heaps differ. Local and dynamic memory on accelerators follow vendor-specific models (device local, shared, unified memory policies) rather than `std::pmr` on the host
Scheduling class	SIMT. Device-side parallel execution model (contrast with host preemption and threading)
Thread-local storage access pattern	Not applicable

P4127R0 [1] Section 9.1 places this row in the bottom group. This row is the **device** environment; host compilation for the same program follows Sections 4.2-4.4.

---

4. Compilers

4.1 Compiler schema

The table lists toolchain families and attributes for comparing implementations against Section 3.

Toolchain	Coro	TLS	Host	Exc	PMR
GCC (`g++`)	Yes	Platform ABI	Both hosted and freestanding	Typically on (hosted)	Yes (hosted libstdc++)
Clang	Yes	Platform ABI	Both hosted and freestanding	Typically on (hosted)	Yes (libc++ / libstdc++)
MSVC	Yes	Windows TLS model	Hosted Windows primary	Typically on	Yes
Arm GNU Toolchain (bare-metal)	Yes with `-fcoroutines` where available	No TLS on typical MCU	Freestanding	Typically off in embedded	No
NVIDIA `nvcc` / host driver	Host side follows host compiler	Host	Host + device split	Follows host	Follows host
EDG C++ front end	Depends on embedding	Depends on embedding	Sold to integrators. No single shipped runtime	Depends on embedding	From host or none

Legend:

- **Coro** - Practical C++20 coroutine support for that row (including embedding-dependent cases).
- **TLS** - `thread_local` code generation or platform caveats at a high level.
- **Host** - Hosted versus freestanding-oriented distribution, or special cases (split host or device, integrator-only front end).
- **Exc** - Exception default for that row. Hosted GCC, Clang, and MSVC match **Typically on** from Section 3; other rows use the same vocabulary where it applies.
- **PMR** - Whether the shipped standard library exposes `<memory_resource>` for hosted builds, or N/A.

4.2 GCC (g++)

Executive summary

GCC is the reference open-source toolchain for many Linux distributions and embedded GNU triples. cppreference lists **GCC 10** as the first release in its C++20 matrix for coroutines (P0912R5) [5].

Schema

Field	Value
Toolchain family	GCC (`g++`)
C++20 coroutine support	Yes
`thread_local` code generation	Platform ABI. Follows the psABI for the selected target (ELF static TLS, PE TLS slots, and so on), not a vendor-specific scheme
Hosted versus freestanding profile	Both hosted and freestanding. One front end with different runtimes: the same `g++` can target a full Linux rootfs or a bare-metal link where only freestanding headers apply
Exceptions (typical default)	Typically on (hosted). Many embedded builds add `-fno-exceptions` explicitly
PMR in shipped standard library	Yes (hosted libstdc++). Complete `<memory_resource>` support from **GCC 9.1** onward [6]

Cross GCC for bare-metal uses the same front end with a runtime link that does not supply hosted libstdc++ for that image (see Section 4.5).

4.3 Clang

Executive summary

Clang is widely used on Darwin, Linux, and Windows. Apple ships a fork tied to Xcode releases.

cppreference lists **Clang 8** as the first column entry for C++20 coroutines (P0912R5 [7]), with partial support in early releases [5]. For **libc++**, polymorphic memory resources (P0220R1 [8]) appear from **Clang 16** in that table [6].

Schema

Field	Value
Toolchain family	Clang
C++20 coroutine support	Yes
`thread_local` code generation	Platform ABI. Same meaning as for GCC: TLS lowering follows the platform binary interface
Hosted versus freestanding profile	Both hosted and freestanding. Also used for bare-metal and kernel targets
Exceptions (typical default)	Typically on (hosted)
PMR in shipped standard library	Yes (libc++ or libstdc++, link-time choice). `<memory_resource>` floors are **Clang 16** with libc++ and **GCC 9.1** libstdc++ in cppreference's table [6]

4.4 MSVC

Executive summary

Microsoft documents C++20 coroutines (P0912R5 [7]) from **Visual Studio 2019 version 16.8** onward (toolset **19.28**) [9]. The Microsoft STL ships `<memory_resource>` (P0220R1 [8]) from **Visual Studio 2017 version 15.6** onward [9].

TLS access patterns for DLLs differ from ELF static TLS as summarised in P4127R0 [1] Section 9.2.

Schema

Field	Value
Toolchain family	MSVC
C++20 coroutine support	Yes
`thread_local` code generation	Windows TLS model. `thread_local` in DLLs can use the slow path described in P4127R0 [1] Section 9.2, unlike a single static link
Hosted versus freestanding profile	Hosted Windows primary. Targets user-mode Windows toolchains with the full CRT, not every experimental or kernel target MSVC can be aimed at
Exceptions (typical default)	Typically on
PMR in shipped standard library	Yes

4.5 Arm GNU Toolchain (bare-metal)

Executive summary

The Arm GNU bare-metal distribution (`arm-none-eabi-gcc`) targets Cortex-M and similar MCUs. Many programs disable exceptions and RTTI to save size.

Schema

Field	Value
Toolchain family	Arm GNU Toolchain (bare-metal)
C++20 coroutine support	Yes with `-fcoroutines` where available. Same floor as upstream GCC (**GCC 10** for P0912R5 [5]). Some triples still need `-fcoroutines` or a newer GCC build; depends on multilib and whether the feature is exposed for that triple
`thread_local` code generation	No TLS on typical MCU. Cortex-M images usually omit thread-local storage because there is no threads model in the same sense as POSIX
Hosted versus freestanding profile	Freestanding
Exceptions (typical default)	Typically off in embedded. The usual `-fno-exceptions` default or project policy for size
PMR in shipped standard library	No

4.6 NVIDIA nvcc (CUDA toolkit)

Executive summary

CUDA compilation splits host code (compiled with a host compiler: GCC, Clang, or MSVC on Windows) and device code processed by `nvcc`. The inventory row records that split: coroutine and PMR questions for application logic refer to the **host** toolchain in the same build.

Schema

Field	Value
Toolchain family	NVIDIA `nvcc` with host driver
C++20 coroutine support	Determined by selected host compiler
`thread_local` code generation	Determined by selected host compiler
Hosted versus freestanding profile	Host environment plus device compilation (split). Device translation units follow NVIDIA's rules for each architecture; see NVIDIA documentation for the supported language subset
Exceptions (typical default)	Determined by host toolchain
PMR in shipped standard library	Determined by host standard library

The build selects a host GCC, Clang, or MSVC for host translation units; version floors follow Sections 4.2-4.4 and cppreference [5] [6].

4.7 EDG C++ front end

Executive summary

Edison Design Group supplies a commercial C++ **front end** (parser and semantic analysis) that compiler vendors embed in their own products [10]. Coroutine support, TLS code generation, exceptions, and PMR are fixed only after the integrator's back end and runtime are chosen. The EDG front end appears in conformance discussions because it tracks the ISO C++ rules closely and is used behind several shipping toolchains.

Schema

Field	Value
Toolchain family	EDG C++ front end
C++20 coroutine support	Depends on embedding
`thread_local` code generation	Depends on embedding
Hosted versus freestanding profile	Sold to integrators. No single shipped runtime or OS target bundled with the front end alone
Exceptions (typical default)	Depends on embedding
PMR in shipped standard library	From host standard library or none. PMR matches the hosted library when one is linked; bare-metal integrations often ship without it

The integrator supplies object layout, TLS lowering, exception machinery, and runtime - attribute runtime behaviour to the **complete toolchain**, not the EDG front end alone. Includes the Comeau C++ compiler; today licensed to vendors building specialised or embedded compilers. For language-conformance questions, committee members still point to EDG-backed builds as a reference parser alongside GCC and Clang.

Acknowledgments

The author thanks Vinnie Falco and the C++ Alliance proposal authors whose platform survey in P4127R0 [1] seeded Section 3.

References

[1] P4127R0 - "The Coroutine Frame Allocator Timing Problem" (Vinnie Falco, C++ Alliance Proposal Team, 2026).

[2] P4100R0 - "The Network Endeavour: Coroutine-Native I/O for C++29" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026).

[3] Freestanding and hosted implementations - cppreference.com overview.

[4] Pigweed pw_async2: Coroutines - Google's embedded C++20 coroutine framework.

[5] Compiler support for C++20 - cppreference.com (per-feature minimum compiler versions, including P0912R5 coroutines).

[6] Compiler support for C++17 - cppreference.com (per-library minimum versions, including P0220R1 `<memory_resource>`).

[7] P0912R5 - "Merge Coroutines TS into C++20 working draft" (Gor Nishanov, 2019).

[8] P0220R1 - "Adopt Library Fundamentals V1 TS Components for C++17 (R1)" (Beman Dawes, 2016).

[9] Microsoft C++ language conformance - Microsoft Learn.

[10] Edison Design Group - Provider of the EDG C++ front end for compiler and tool vendors.