



How to use the HSP for executing DSP and CNN operations on STM32 MCUs

Introduction

This application note applies to STM32 microcontrollers featuring the Hardware Signal Processor (HSP) subsystem as listed in [Table 1](#).

The HSP is a dedicated signal-processing engine designed to accelerate complex mathematical operations and reduce CPU load in real-time applications. It combines a high-performance floating-point core with event-driven architecture, enabling deterministic execution of signal-processing chains.

The HSP is particularly useful in domains such as digital power control, energy metering, motor control, and audio processing, where algorithms require fast, predictable computation and efficient data handling. On supported devices, HSP can also execute small CNN kernels for AI workloads, complementing traditional DSP tasks.

This document provides an overview of the HSP architecture, its operating modes, and its integration within STM32 platforms. It explains how to leverage the HSP for high-performance DSP and AI tasks, and it includes practical examples on DSP and CNN applications, with performance measurements, optimization tips, and comparisons against software implementations.

The code examples referenced in this note were tested on the STM32 [NUCLEO-U3C5ZI-Q](#) board.

Table 1. Applicable products

Type	products
Microcontroller	STM32U3B5/3C5 line

1 General information

This document applies to the Arm® Cortex® core-based STM32 microcontrollers.



Note:

Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere.

The Arm word and logo are trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved.

Reference documents

To fully benefit from this application note, the reader must be familiar with the following documents, which are available at www.st.com.

- [1] STM32U3 series Arm®-based 32-bit MCUs (RM0487)
- [2] HSP processing functions on STM32 MCUs (UM3360)
- [3] STM32U3 series datasheets

2 HSP introduction

2.1 What the HSP is and why it exists

The hardware signal processor (HSP) is a dedicated, on-chip signal-processing subsystem that offloads intensive algorithms from the main CPU to a specialized signal processor engine (SPE).

By transferring heavy digital signal processing (DSP), applications achieve deterministic latency, higher throughput, and lower CPU and system power consumption. This approach preserves CPU bandwidth for control, input/output (I/O), communication, and other tasks.

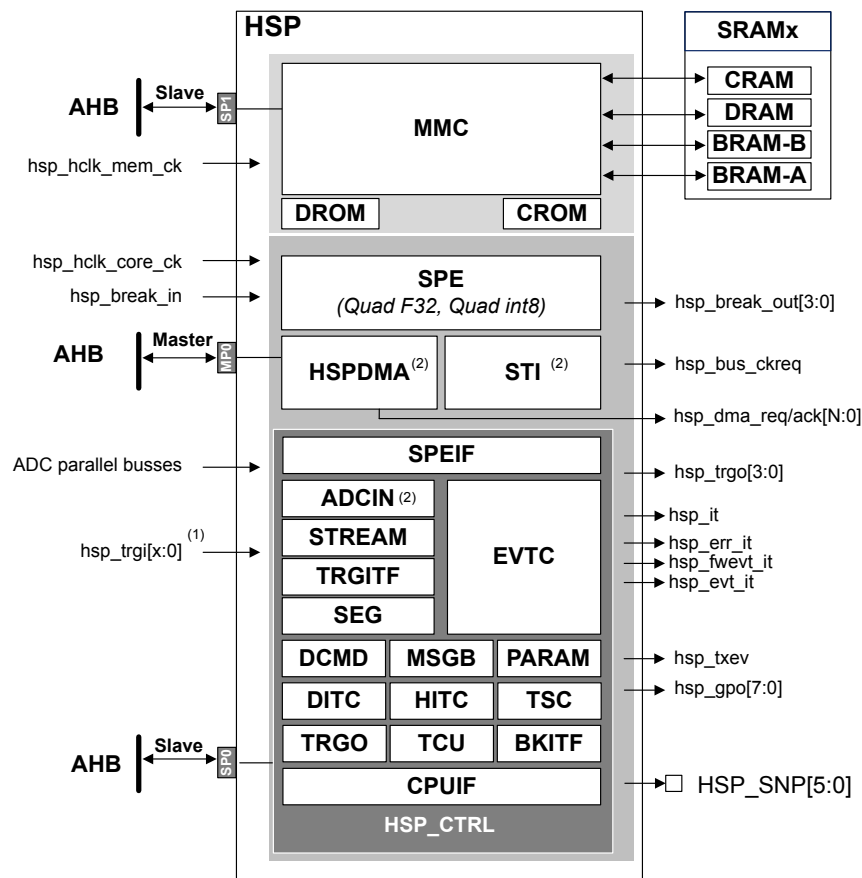
The HSP is tightly coupled to on-chip memories and to an event-driven scheduler. Therefore, the HSP can react to external triggers, stream data from peripherals, and run complex processing chains autonomously.

Typical use cases include digital power control, energy metering, motor control, and audio processing, where real-time behavior and consistent cycle budgets matter. On supported devices, the HSP operates as a small AI accelerator, running CNN layers. Supported layers are quantized int8 point wise (PW), depth wise (DW), Conv2D, 2D pooling, and fully connected/dense layers.

2.2 Overview of HSP architecture

The hardware signal processor (HSP) is organized as a subsystem that combines a high-performance engine with dedicated memory resources, event-driven scheduling, and flexible data interfaces. Its architecture is designed to deliver deterministic execution of signal-processing tasks while minimizing CPU involvement and system latency.

Figure 1. HSP block diagram



DT80421V2

1. The number of trigger inputs is 20 for the STM32U3B5/3C5 line.
2. This feature is not supported on all devices; therefore, developers should confirm product capabilities before relying on it.

The block diagram in [Figure 1](#) illustrates how each element interconnects from the HSP to the MCU. Each block plays a specific role in enabling autonomous operation, low-latency command execution, and robust debug capabilities:

- Signal processor engine (SPE). A single-precision (float32) DSP core executes microcoded kernels in CROM plus the application firmware loaded into CRAM. It reads/writes operands in DRAM and BRAM and interacts with control/status registers exposed to the CPU.
- Private memories and matrix. Program (CROM/CRAM) and data (DROM/DRAM) memories are tightly coupled to the SPE for bandwidth and predictability. An internal memory controller (MMC) arbitrates between SPE and AHB initiators.
- The event controller (EVTC) supports various sources. These sources include stream buffer (STREAM) and matrixed trigger inputs (TRGIN) from the SoC. They also include software events from the CPU/SPE and direct command (DCMD) requests.

Interfaces for data and control:

- The STREAM interface transfers words between the system-on-chip (SoC) and the HSP. It uses ready and empty flags, and it detects overrun and underrun conditions. The interface can convert integers to floating-point values for inbound data. Use the STREAM interface to trigger HSP processing when the application provides data to this interface or reads data from it. The interface allows the application to configure the processing source.
- Trigger-out (TRGO) generates one-cycle pulses when it reaches internal milestones. These milestones include task transitions and timestamp captures. This feature helps to synchronize DMA and timers.

Observability, safety, and debug high level. A full detailed explanation of debug features available on the HSP is presented in [Section 5](#).

- Timestamp capture (TSC) records transitions of task or processing functions. It includes an optional free-running timestamp. TSC can also raise a trigger output (TRGO) pulse. This action allows a SoC DMA to capture the register without needing CPU polling.
- The task comparator unit (TCU) detects task start/end and overlap.
- The break interface (BKITF) can stop the SPE on errors. It can stop at chosen processing function markers. It can also stop at the start of a target processing list, or at the next processing function call. Additionally, it can coordinate with CPU debug.
- A comprehensive interruption tree exposes granular status, clear and enable bits, and FPU errors.

Separate memory and core clocks allow the MCU to access HSP memories while the SPE is idle.

2.3 Operating modes

The hardware signal processor (HSP) offers two distinct operating modes that define how signal-processing tasks are scheduled and executed: the sequencer mode and the accelerator mode. These modes are designed to address different application needs, from highly deterministic operations to flexible, on-demand applications.

In sequencer mode, the HSP operates as an autonomous engine capable of executing prerecorded processing lists with minimal intervention from the host CPU. This mode is ideal for periodic or continuous signal-processing chains where timing predictability and low overhead are critical.

In accelerator mode, the HSP acts as a callable function accelerator, executing individual signal processing functions or CNN layers on demand. This mode allows an application to accelerate the execution of a task processing in an opportunistic manner without significant porting effort.

Each mode provides identical performance results because both use the same hardware engine for computation. The user can select the mode that best fits the application requirements, or combine both modes within the same application to maximize flexibility.

These sections detail the sequencer and accelerator modes.

- Sequencer mode (processing lists)
 - The user builds and records processing lists (PLs) that describe a dedicated algorithm as an ordered set of processing functions (PFs) (for example, FIR, IIR, FFT, vector math, etc.).
 - During recording, the firmware generates the PL as SPE-native code. It loads this code into CRAM and emits a parameter table into DRAM. At runtime, the event controller selects the PL task, and the SPE jumps directly to the compiled entry point. This mode is ideal for autonomous pipelines and deterministic, periodic processing, where you want fixed timing and minimal control overhead.
- Accelerator mode (direct command)
 - The CPU (or another controller) issues a direct command (DCMD) with a function ID and up to three live pointers/arguments. The hardware raises a busy flag and triggers a reserved task. The SPE executes the requested function with very low latency.
 - All parameters can vary call-by-call; there is no need to prerecord a processing list. This is the simplest way to start DSP operations, or drive CNN layers on supported parts.

On many implementations, CNN functions are accelerator-only (not recorded as PLs) and use a dedicated BRAM-A/B workspace layout.

2.4 Features

2.4.1 Event-driven execution

The event controller (EVTC) consolidates asynchronous and synchronous sources: STREAM interface when buffers become ready or empty, TRGIN edges selected from the SoC trigger matrix, software events posted by CPU/SPE (shared and dedicated), and DCMD requests. A fixed priority scheme ensures that urgent work (for example, high-rate acquisition) preempts lower-priority tasks.

Once an event is selected, the task controller checks the task number in the table preloaded by the firmware. It then wakes the SPE and performs a direct jump to the task entry. This table reduces as much as possible the latency between the activation of an event and the execution of a task.

When the HSP starts the execution of a task, even a task with higher priority cannot interrupt the ongoing task. The pending events are held until the end of the current task, then the priority encoder selects the one with the highest priority and executes the corresponding processing list.

2.4.2 STREAM interface

Each buffer is configured C2H (CPU→HSP, read by SPE) or H2C (HSP→CPU, written by SPE). Ready/empty conditions raise flags; the engine holds the event until the buffer is serviced (read or written). Optional integer to float (I2F) conversion turns incoming integers into float32 before the SPE computation, simplifying front end integration. Simple combination logic can merge flags, so a set of related buffers is serviced as one event. Overrun/underrun conditions are latched for diagnostics.

2.4.3 Signal processing engine (SPE)

The SPE is the core compute unit of the HSP. It executes DSP and CNN kernels using microcoded instructions stored in CROM and CRAM. Operating on single-precision floating-point data, the SPE performs arithmetic-intensive operations such as FIR filtering, FFT, and matrix computations with deterministic timing. It accesses operands directly from BRAM for maximum bandwidth and uses DRAM for parameter tables.

2.4.4 Memory controller (MMC)

The MMC manages arbitration between the SPE and AHB initiators such as the CPU and DMA. It ensures priority for the SPE during real-time processing while providing controlled access slots for the CPU. This mechanism ensures deterministic bandwidth and prevents contention.

2.4.5 Event controller (EVTC)

The EVTC is responsible for event-driven scheduling. It consolidates triggers from multiple sources: STREAM buffer flags, TRGIN signals, software events, and DCMD requests and maps them to tasks using a firmware table. Once an event is selected, the EVTC wakes the SPE and jumps to the corresponding entry point, enabling a low-latency response to asynchronous events.

2.4.6 Trigger input (TRGIN)

The TRGIN allows external synchronization of HSP tasks with SoC peripherals such as timers or ADC end-of-conversion signals. This capability ensures deterministic alignment of DSP execution with system-level events, which is critical for control loops and real-time acquisition.

2.4.7 Analog-to-digital converter input (ADCIN)

The ADCIN provides a direct path for feeding raw ADC samples into the HSP without CPU intervention. This reduces latency and eliminates unnecessary memory copies, enabling real-time DSP pipelines for acquisition and processing.

Note: The ADCIN is not available on all devices, so developers should verify product capabilities before relying on this feature.

2.4.8 Hardware signal processor DMA (HSPDMA)

On some products, the HSP includes a dedicated DMA engine—HSPDMA—for autonomous data transfers between BRAM and system memory or peripherals. This feature offloads the CPU entirely during data movement and optimizes throughput for continuous streaming applications.

Note: The HSPDMA is not available on all devices, so developers should verify product capabilities before relying on this feature.

2.5 Data types and formats (DSP and CNN)

This section describes the data types and formats supported by the HSP. It also covers the memory layouts used for vectors, matrices, and complex data. Finally, it outlines the pointer models that define how algorithms use their operands.

Understanding these formats is essential to build processing lists, configure direct commands, and optimize data transfers between the HSP and the rest of the system.

Numeric domains (DSP paths)

- Float32 (IEEE-754 single precision) is the native compute type for processing functions (filters, transforms, vector ops, etc.).
- Integers (fixed-point) can be supplied on streams buffers; the HSP offers integer-to-float conversion on the fly. Implementations accept signed integers up to 32-bit and unsigned up to 31-bit for I2F conversion.
- Complex numbers are stored as interleaved float32 pairs in the order: real part, imaginary part, and at ascending addresses for vector and FFT workloads.
- Matrices are typically stored row-major (float32)

CNN domain (on supported devices)

- Middleware supports only quantized int8 layers. Tensors for inputs/outputs and filter coefficients are represented in int8 while bias and scaling factors are represented in int32.
- Working buffers live in BRAM-A/B and following a ping-pong/circular pattern depending on the layer.
- CNN operators usually run in accelerator mode (DCMD). They do not record as PLs. They can coexist with DSP PLs if the CNN has exclusive access to its resources during execution.

Memory access granularity

- BRAM supports 64-bit SPE loads/stores for bandwidth (handy for complex pairs) and 32/16/8-bit accesses where needed.
- STREAM buffers are 32-bit word addressed on the CPU side; the I2F stage expands to float32 for the SPE.
- PARAM registers are 32-bit scalars used for quick arguments.

To keep processing code tight and deterministic, the HSP uses pointer descriptors that tell the firmware where operands reside and whether any indirection is needed:

- ABMP (BRAM-AB pointer): direct pointer into the BRAM-AB working area; best performance for hot data (states, taps, vectors).
- POP (pointer-to-pointer): a BRAM location that contains a pointer; lets software retarget buffers at runtime without rerecording PLs.
- IBP (interface buffer pointer): operands residing in interface FIFOs (for example, ADCIN or STREAM buffer) when the hardware path is used.
- IMM (immediate): small constants embedded in the parameter block.

In sequencer mode, a compact IOTYPE field indicates the pointer form for each operand. It also shows how to bind it efficiently (for example, follow POP, set up STI-AP, and read IBP).

In accelerator mode, IOTYPE is not required; the SPE can infer whether a pointer is in BRAM-AB or global by its address range, which keeps the call overhead extremely low.

2.6 BRAM organization

The buffer RAM (BRAM) is the working memory of the HSP and plays a critical role in achieving deterministic performance. BRAM is organized as two separate banks of 16 KB: BRAM_A and BRAM_B that can be accessed in two different ways.

- Independent access: BRAM_A and BRAM_B have their dedicated address space. Each bank (A and B) is independent and can be allocated separately. This addressing mode is used exclusively by CNN middleware functions. BRAM_A and BRAM_B can carry different data.
- Dual-port RAM access: BRAM_AB represents a shared memory space for BRAM_A and BRAM_B. A write access results in the same data being copied to both physical BRAM_A and BRAM_B banks. Data used in DSP accelerator mode or sequencer mode is allocated and addressed using this mode. BRAM_A and BRAM_B carry the same data (duplicated).

Note: *BRAM_AB must be used for all HSP operations except CNN operations because it supports concurrent access and offers the highest flexibility for streaming and intermediate buffers.*

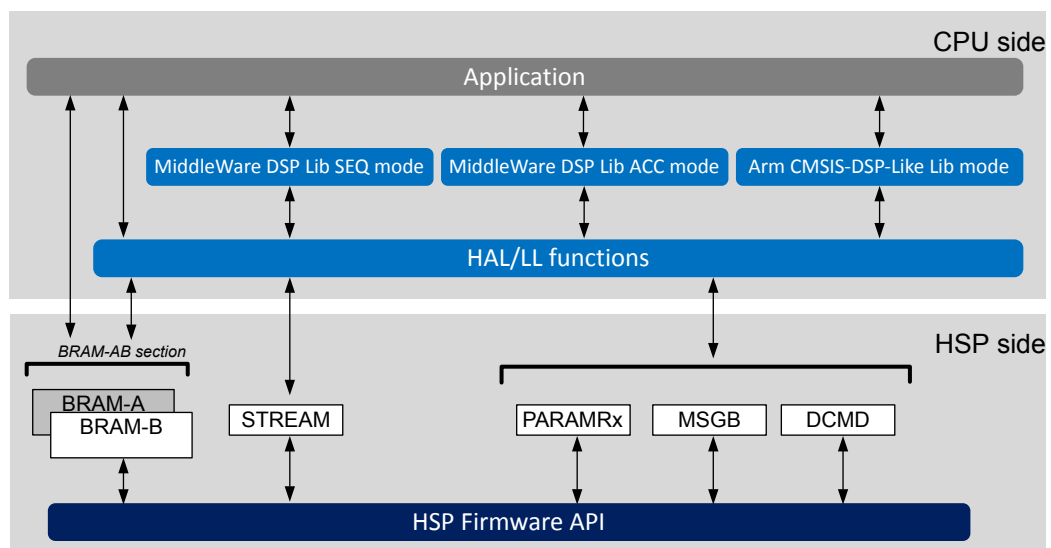
Alignment and granularity are essential for performance. Scalar data uses 32-bit alignment, with support for 16-bit and 8-bit when required. This alignment ensures efficient burst transfers and avoids penalties during computation.

2.7 BRAM Allocation strategies by use case

2.7.1 DSP-only scenario

In DSP-centric applications, BRAM-AB is allocated for all working buffers, including filter states, FIR/IIR taps, and FFT vectors. Buffers are typically organized as ping-pong or circular structures to support continuous streaming through the STREAM interface. This arrangement minimizes latency and ensures smooth data flow between the CPU and the HSP.

Figure 2. Data sharing models for standard DSP functions

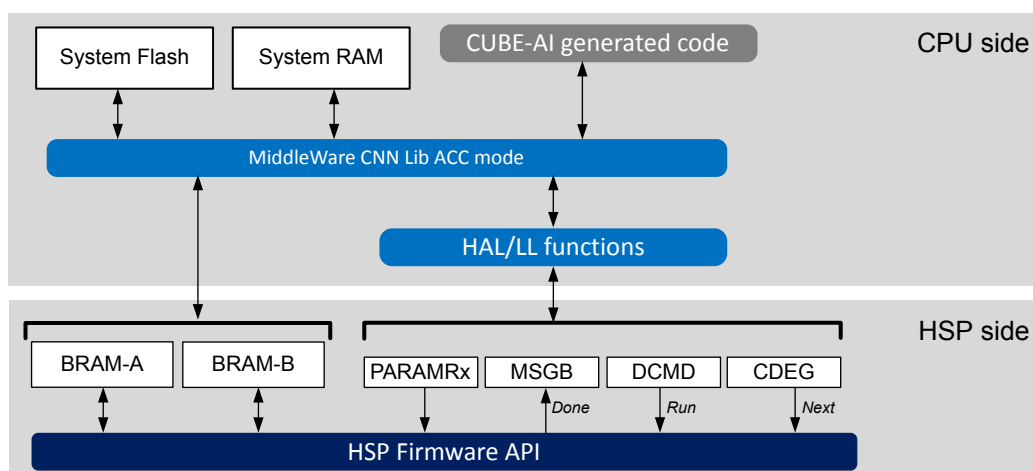


DT77845V1

2.7.2 CNN-only scenario

For CNN workloads, BRAM-A and BRAM-B are reserved exclusively. Each layer requires a well-defined layout that includes the input/output data, kernel weights, bias values, and quantization tables. The middleware manages these allocations automatically, ensuring that memory is aligned and sized correctly for each layer.

Note: *The output, input, and kernel data format are int8.*

Figure 3. Data sharing models for standard CNN functions


DT77818V1

2.7.3 Mixed DSP and CNN scenario

When DSP and CNN tasks coexist, strict rules apply. CNN execution requires BRAM-A/B, while DSP uses BRAM-AB. During CNN processing, non-CNN events must be disabled to avoid interference. The middleware reserves memory regions upfront to prevent overlapping and fragmentation, ensuring both workloads can run safely without corrupting data. The BRAM can be split into two areas:

- Persistent area: No CNN processing can access this. This area is addressed in BRAM_AB mode. It carries DSP data that should remain persistent, such as fixed coefficients or memories.
- Nonpersistent area (default): DSP functions can use it in BRAM_AB mode, or CNN functions can use it in BRAM_A or BRAM_B mode. This is an overlay area that both DSP and CNN components can have access to.

3 HSP middleware overview

3.1 Introduction

The HSP middleware serves as an abstraction layer between the application code and the HSP hardware. Its purpose is to make the hardware signal processor easier to use. It achieves this by hiding the details of low-level register programming and memory management behind a high-level API. This approach allows developers to focus on algorithm design and application logic rather than hardware-specific constraints.

The middleware plays a critical role in configuration, managing memory resources, and organizing task scheduling. It offers services to start the HSP. It loads firmware, allocates buffers, and maps them. It also manages event-driven execution. The developer does not need to handle complex hardware registers directly. By doing so, it ensures portability and reduces development time while maintaining deterministic performance.

Integration with the STM32Cube ecosystem is a base of the middleware design. It is built on top of the STM32 hardware abstraction layer (HAL), ensuring consistency with other STM32 peripherals and enabling coexistence with [STM32CubeMX](#) configuration tools. This integration allows developers to configure the HSP alongside other system components in a unified environment and generate initialization code automatically.

In summary, the HSP middleware connects high-level DSP or AI application requirements to low-level HSP hardware capabilities. It provides robust and maintainable solutions on STM32 microcontrollers.

3.2 Middleware architecture

The middleware is organized into four main components. Each component manages a specific phase of the HSP life cycle.

3.2.1 Components

Initialization Layer

This layer handles the boot sequence of the HSP. It powers up the subsystem, enables clocks, initializes memory, and loads the firmware into CRAM and DRAM. It also performs integrity checks and applies resource locks to protect code and data once initialization is complete. The initialization layer ensures the HSP starts in a clean state and is ready for execution.

Configuration Layer

Once the HSP is active, the configuration layer sets up the processing environment. In sequencer mode, it records processing lists (PLs) by compiling them into CRAM and generating parameter tables in DRAM. In accelerator mode, it prepares direct command (DCMD) interfaces for low-latency calls. This layer also configures STREAM buffers, trigger routing, and event priorities. It ensures that data flows and task scheduling match the timing requirements of the application.

Execution Layer

This layer manages task dispatch and event handling during runtime. It monitors event sources such as STREAM flags, TRGIN triggers, and software events, and ensures that the correct task is activated in response. It also supervises the interaction between the CPU and the HSP, handling mailbox messages and parameter updates. The execution layer ensures that the HSP runs autonomously while maintaining synchronization with the rest of the system.

Debug and profiling APIs

Integrated APIs allow developers to trace execution and measure performance without disrupting real-time behavior. These include timestamp capture (TSC) for cycle-accurate timing, task comparator unit (TCU) for monitoring tasks, and break interface (BKITF) for controlled halts during debugging. Middleware exposes these features through APIs so developers can enable profiling or set breakpoints without manual register manipulation.

3.2.2 Interaction with system DMA, interrupt services, and STEdgeAI-Core for CNN

The middleware coordinates with DMA controllers to offload data transfers between HSP memories and system RAM or peripherals. For example, DMA automatically drains STREAM buffers or captures logs from the TSC without CPU intervention. This process reduces latency and power consumption.

Interrupt services are abstracted so that application code can respond to HSP events (such as task completion or error conditions) through callbacks rather than polling.

The middleware offers direct functions for AI workloads. It can run a complete single CNN layer on the HSP. This includes quantized int8 point wise, depth wise, fully connected, convolutional 2D, or pooling. It manages BRAM-A/B allocation, configures DCMD calls for each layer, and synchronizes execution with the CPU to ensure correct sequencing of AI pipelines. Those CNN middleware functions can be called directly from the application code. It is much easier to use [STEdgeAI-Core](#). It produces code that calls the best CNN middleware function. If the HSP cannot support a specific layer, fall back to an STM32 implementation instead.

3.3 Middleware services for memory management

The middleware provides a set of services to simplify memory handling and reduce the complexity of manual address management:

BRAM allocation and mapping

APIs enable developers to reserve BRAM regions and return pointers in the correct SPE address space. For example, ABMP is used for direct BRAM pointers, while POP supports indirect pointers for dynamic retargeting. Additional features include handling IOTYPE descriptors in sequencer mode. These descriptors encode operand types for efficient binding. The system also supports dynamic pointer updates for POP and guarded memory protection (GMP) operands. This support enables runtime flexibility without re-recording PLs.

3.4 Middleware options for performance and debug

Performance tuning and debugging are integrated into the middleware through dedicated options:

- **TSC integration:** APIs enable timestamp capture for cycle logging. This is essential for building performance tables and validating timing constraints.
- **BKITF control macros:** Developers can set breakpoints at the start of a processing list. They can also set breakpoints before a specific function or during error conditions. This process does not require writing registers manually.
- **Other utilities:** Functions are available to read cycle counters, monitor DMA throughput, and compute effective processing rates. These tools help developers optimize memory usage and scheduling.

HSP execution is predictable; however, it can vary when BRAM-AB is shared with other bus masters. Under heavy contention, throughput may degrade depending on arbitration settings (BARB) and traffic density. Typical worst-case values are approximately 6% (BARB = 0), 12% (BARB = 1), and 25% (BARB = 2) when an external master continuously requests BRAM bandwidth. Real applications rarely reach these extremes.

When a PF uses a pointer-to-pointer (POP), the value is refreshed before execution, which adds approximately four cycles per POP. Use POP only when dynamic retargeting is required.

4 Code examples and performance

This section provides practical examples of how to use the HSP for DSP and CNN operations, using the Middleware functions. It is designed for developers who want to understand not only how to implement these features but also how to evaluate their performance.

The examples are organized into three categories:

- DSP Examples: Showcases filtering and transformation operations in both Sequencer and Accelerator modes.
- CNN Examples.
- Mixed DSP + CNN: Explores hybrid use cases where DSP preprocessing feeds CNN inference.

For each example, the section:

- Present clear code snippets.
- Explain configuration steps.
- Provide performance metrics.
- Discuss trade-offs and optimization tips.

4.1 DSP examples

4.1.1 Sequencer mode

This example illustrates how the HSP operates in sequencer mode with full middleware support, making hardware interaction almost transparent to the user.

The pipeline includes:

- Windowing
- FIR filtering (64 NTAPS)
- FFT computation (IFFT = 0, BITREV = 1)
- Power spectrum calculation

Once configured, the HSP executes autonomously with minimal user involvement.

The middleware abstracts all low-level details such as:

- HSP initialization
- BRAM allocation and memory mapping
- Filter state management
- Instructions include FIR, FFT, and other primitives.

Execution flow (middleware-driven)

1. Initialization

```
HSP_Engine_IF_Init(&hmw)
```

Perform a hardware-level reset of the HSP block. This action clears any residual configuration, stale flags, or partially recorded lists.

```
HSP_CORE_Init(&hmw, hmw.hdriver)
```

Initialize the HSP in sequencer mode. Load the HSP firmware (FW) into CRAM and DRAM. Then, boot the HSP engine.

```
float32_t *PP_Buffer = (float32_t *)HSP_BRAM_Malloc(&hmw, (MAX_BLOCKSIZE * 2),
                                                    HSP_BRAM_ALLOCATION_DEFAULT);
float32_t *Window = (float32_t *)HSP_BRAM_Malloc(&hmw, MAX_BLOCKSIZE,
                                                  HSP_BRAM_ALLOCATION_DEFAULT);
float32_t *coef_Fir = (float32_t *)HSP_BRAM_Malloc(&hmw, FIR_NB_COEF,
                                                  HSP_BRAM_ALLOCATION_DEFAULT);
```

The middleware allocates into the BRAM-AB the space for the working variables.

```
uint32_t Flt1 = HSP_BRAM_MallocStateBuffer_Fir(&hmw, FIR_NB_COEF, WIND_SIZE, HSP_BRAM_ALLO
CATION_DEFAULT);
```

Create a state buffer for the FIR filter in BRAM-AB.

```
memcpy((uint32_t *) Coef_Flt1_64ntaps, (uint32_t *)coef_Fir, FIR_NB_COEF);
```

Load the coefficient into the BRAM-AB.

```
HAL_HSP_TCU_StartComparator(&hhspl, HAL_HSP_TASK_COMPARATOR_0, (HAL_HSP_TaskTypeDef)PL8, HA
L_HSP_IT_EVT_TCU0_TASK_ENDED);
```

Configure comparator 0 on the end of task of the processing list 8.

2. Processing list creation

```
// Configure processing list PL8 with conditional execution and FFT
HSP_SEQ_StartRecordPL(&hmw, PL8); // Start recording the processing list
HSP_SEQ_IfIFloat(&hmw, HSP_SEQ_CMP_IFEQ, PP_value, 0.0); // IF PP_value == 0.0

// ping buffer processing
HSP_SEQ_ScaISet_f32(&hmw, 1.0, PP_value, HSP_SEQ_IOTYPE_DEFAULT); // PP_value = 1.0
HSP_SEQ_Fir_f32(&hmw, PP_Buffer, coef_Fir, Flt1, PP_Buffer, MAX_BLOCKSIZE, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_VectMul_f32(&hmw, PP_Buffer, Window, PP_Buffer, MAX_BLOCKSIZE, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_Rfft_f32(&hmw, PP_Buffer, (hsp_ftt_lognbp_cmd_t)10, (uint8_t)RFFT, (uint8_t)BITREV,
                HSP_RFFT_TYPE_1, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_CmplxMagSquared_f32(&hmw, PP_Buffer, PP_Buffer, (MAX_BLOCKSIZE >> 1), HSP_SEQ_IOTYPE_DEFAULT);

HSP_SEQ_IfElse(&hmw); // ELSE branch

// pong buffer processing
HSP_SEQ_ScaISet_f32(&hmw, 0.0, PP_value, HSP_SEQ_IOTYPE_DEFAULT); // PP_value = 0.0 (Pong buffer
// selection)
HSP_SEQ_Fir_f32(&hmw, &PP_Buffer[MAX_BLOCKSIZE], coef_Fir, Flt1, &PP_Buffer[MAX_BLOCKSIZE],
                MAX_BLOCKSIZE, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_VectMul_f32(&hmw, &PP_Buffer[MAX_BLOCKSIZE], Window, &PP_Buffer[MAX_BLOCKSIZE],
                MAX_BLOCKSIZE, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_Rfft_f32(&hmw, &PP_Buffer[MAX_BLOCKSIZE], (hsp_ftt_lognbp_cmd_t)10, (uint8_t)RFFT,
                (uint8_t)BITREV, HSP_RFFT_TYPE_1, HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_CmplxMagSquared_f32(&hmw, &PP_Buffer[MAX_BLOCKSIZE], &PP_Buffer[MAX_BLOCKSIZE],
                (MAX_BLOCKSIZE >> 1), HSP_SEQ_IOTYPE_DEFAULT);
HSP_SEQ_IfEndif(&hmw); // End IF / ELSE block

HSP_SEQ_SetFlag(&hmw, TASK_DONE); // Set TASK_DONE flag when processing list is finished
HSP_SEQ_StopRecordPL(&hmw); // Stop recording the processing list PL8
```

- HSP_SEQ_StartRecordP() and ...End() delimit the recording phase of the processing list number 8 (PL8).
- DSP primitives (FIR, FFT, etc.) are added via simple API calls.
- Conditional branching using HSP_SEQ_IfIFloat and HSP_SEQ_IfElse for ping-pong buffers is handled internally by the middleware.

3. Execution phase

```
HSP_SEQ_RunTask(&hmv, PL8, TIMEOUT_VALUE)
```

The middleware triggers the PL by writing its ID through a single API call. The middleware signals completion by managing flags.

4. Performance

Figure 4. Acceleration ratio between HSP in sequencer mode and Arm® Cortex®-M33

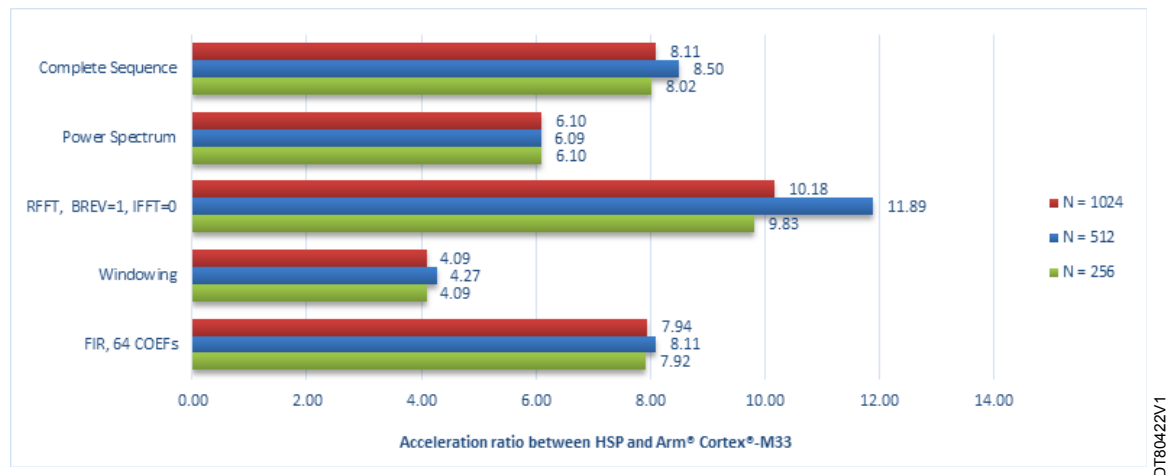
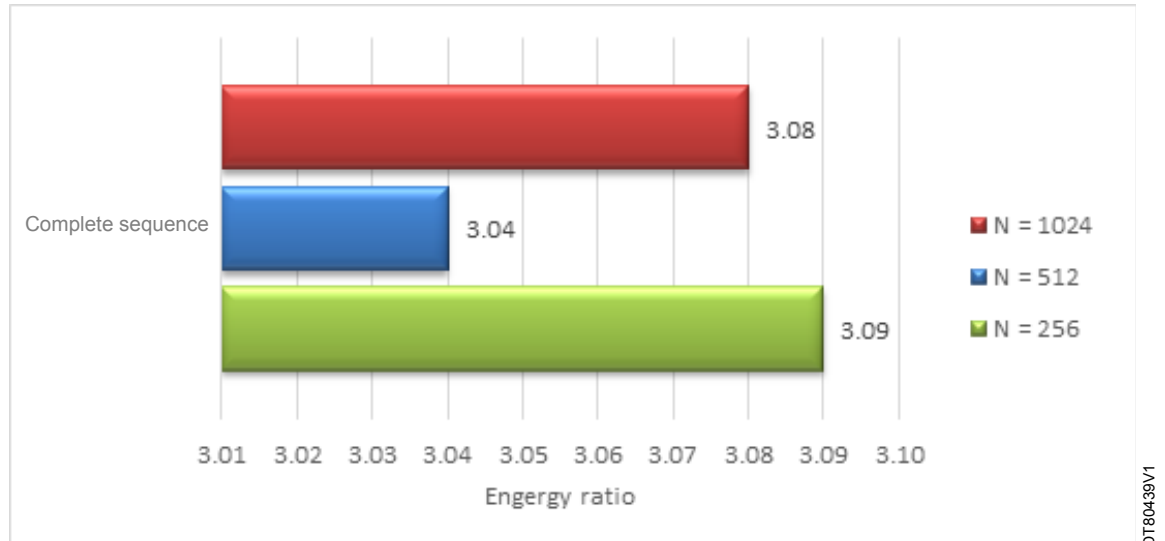


Figure 5. Energy gain ratio between HSP in sequencer mode and Arm® Cortex®-M33



The HSP in sequence mode provides high performance and energy savings compared with an Arm® Cortex®-M33 software implementation.

For the tested block sizes, N is the number of samples per block. The values are N = 256, N = 512, and N = 1024. Computation occurs on 32-bit floating-point data.

The complete DSP pipeline, including windowing, FIR filtering, FFT, and power spectrum calculation, runs approximately eight times faster in sequencer mode than in an Arm® Cortex®-M33 software implementation.

Within this chain, the FFT step alone is approximately 12 times faster.

Overall energy consumption is reduced by about three times, so the same result is obtained much faster while using significantly less power.

Sequencer mode, combined with middleware-driven configuration, offers high throughput, low latency, and significant energy savings. This combination is suitable for real-time and power-sensitive applications.

4.1.2 Accelerator mode

Accelerator mode enables execution of DSP primitives on demand without recording a full processing list. This example describes a three-step pipeline:

- Windowing: Apply a window function to the input data.
- Complex FFT (CFFT): Compute the frequency spectrum.
- Power spectrum: Calculate the square magnitude of FFT output.

This sequence demonstrates how accelerator mode can chain multiple DSP operations dynamically, without recording them as a processing list. Each function call is independent and uses CMSIS-like APIs, making the code familiar and easy to port from existing Arm® CMSIS-DSP implementations.

The benefits of this mode are:

- Immediate execution: No need to record sequences.
- Dynamic workload: Perfect for real-time systems where operations change frequently.
- CMSIS-like APIs: The HSP middleware mimics Arm® CMSIS-DSP function signatures, so porting is straightforward.

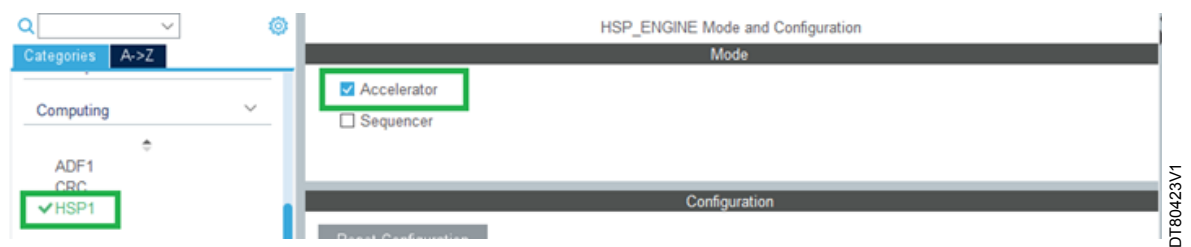
If the original application already uses CMSIS-DSP, swap `arm_*` calls for `stm32_hsp_*` equivalents and gain hardware acceleration with minimal changes.

Execution flow (middleware-driven)

1. Initialization

Before calling any DSP function, initialize the HSP: In **STM32CubeMX**, under the computing category, select the HSP1 instance and activate the accelerator mode.

Figure 6. STM32CubeMX settings: enable HSP accelerator mode



This procedure automatically generates the call to initialize the HSP handlers in the application:

```
if (HSP_Engine_IF_Init(&hwm) != HSP_IF_OK)
if (HSP_CORE_Init(&hwm, hwm.hdriver) != HSP_CORE_OK) /* To lock the HSP, */
/* a hardware reset is required */
if (HSP_CORE_Lock(&hwm) != HSP_CORE_OK)
```

These calls ensure that the HSP firmware is loaded, clocks are enabled, and the engine is ready.

2. Allocate buffers

Working buffers must reside in BRAM-AB for best performance.

```
float32_t *PP_Buffer = (float32_t *)HSP_BRAM_Malloc(&hwm, (MAX_BLOCKSIZE * 2),
HSP_BRAM_ALLOCATION_DEFAULT);
float32_t *Window = (float32_t *)HSP_BRAM_Malloc(&hwm, MAX_BLOCKSIZE,
HSP_BRAM_ALLOCATION_DEFAULT);
```

The middleware handles BRAM allocation, so you do not need to compute offsets or translate addresses manually.

3. Create and initialize instances

Prepare CMSIS-like instance. For CFFT, initialize the instance using `stm32_hsp_*_init_*` APIs.

```
stm32_hsp_cfft_instance_f32 Acc_Cfft;
stm32_hsp_cfft_init_f32(&Acc_Cfft, MAX_BLOCKSIZE);
```

Note: Signatures match Arm® CMSIS (`arm_cfft_init_f32`), but HSP simplifies memory management.

4. Call accelerator functions

Invoke DSP primitives (windowing, FFT, magnitude) using CMSIS-style signatures.

```
// Step 1: Apply window (element-wise multiply)
stm32_hsp_mult_f32(PP_Buffer, Window, PP_Buffer, MAX_BLOCKSIZE);
// Step 2: Perform complex FFT
stm32_hsp_cfft_f32(&Acc_Cfft, PP_Buffer, IFFT, BITREV);
// Step 3: Compute power spectrum (magnitude squared)
stm32_hsp_cmplx_mag_squared_f32(PP_Buffer, PP_Buffer, (MAX_BLOCKSIZE >> 1));
```

Note:

Each function mirrors the semantics of the CMSIS-DSP library but executes on hardware, enabling up to 10 times faster execution.

5. CMSIS-DSP versus HSP mapping

There is no difference in usage between CMSIS-DSP and HSP in accelerator mode. The middleware has been designed to fully support the same instance structures and function signatures as CMSIS-DSP, so developers can reuse existing code with minimal changes. Table 2 lists the CMSIS-DSP functions and their equivalents in the HSP.

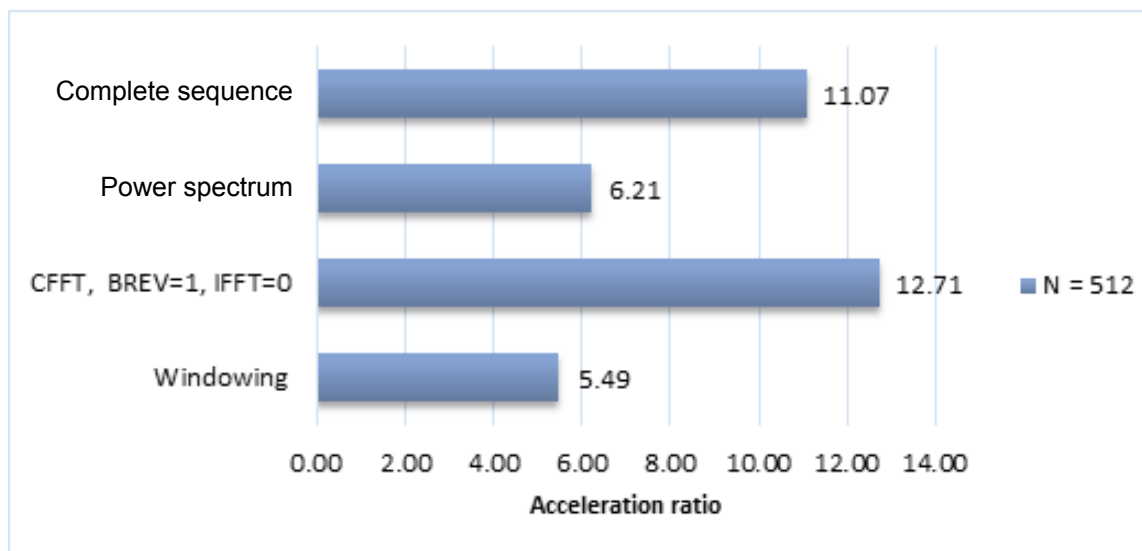
Table 2. CMSIS-DSP versus HSP mapping

CMSIS-DSP function	HSP accelerator equivalent
arm_cfft_init_f32	stm32_hsp_cfft_init_f32
arm_mult_f32	stm32_hsp_mult_f32
arm_cfft_f32	stm32_hsp_cfft_f32
arm_cmplx_mag_squared_f32	stm32_hsp_cmplx_mag_squared_f32

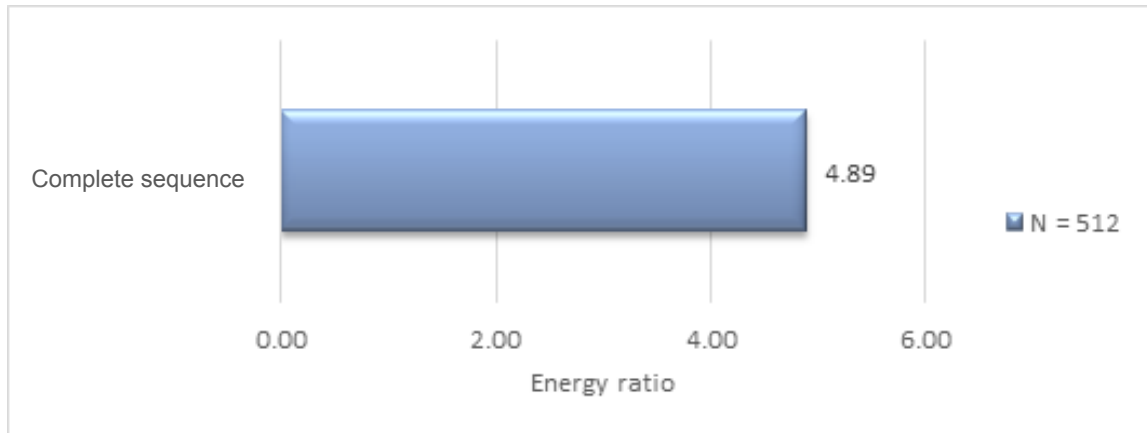
6. Performance

In accelerator mode, the HSP delivers outstanding performance compared with an Arm® Cortex®-M33 when executing DSP primitives individually. Figure 7 illustrates the acceleration ratio for three key operations: windowing, complex FFT (CFFT), and power spectrum as well as the complete sequence that combines these steps, for a block size of N = 512 samples. Computation occurs on 32-bit floating-point data.

Figure 7. Acceleration ratio between HSP in accelerator mode and Arm® Cortex®-M33



DT80424V1

Figure 8. Energy gain ratio between HSP in accelerator mode and Arm® Cortex®-M33


DT80425V1

For STM32 users, the HSP provides a significant increase in efficiency. It executes windowing, FFT, and power spectrum processing 5 to 12.7 times faster, and reduces energy consumption by a factor of 5 compared to a standard Arm® Cortex®-M33 implementation.

Although calls are blocking, the execution time is significantly reduced. Therefore, the CPU wait time is negligible compared to the performance benefit. Accelerator mode provides flexibility and simplicity, and delivers double-digit performance gains. This characteristic makes accelerator mode suitable for dynamic digital signal processing workloads.

For cycle budgets and conditions per function, including filters, transforms, vector and matrix operations, and complex operations, see [Section 7](#).

4.1.3 Optimization tips

To maximize performance and minimize energy consumption when running DSP workloads in sequencer mode, apply several strategies. These optimizations use both hardware capabilities and middleware features to ensure that the HSP operates at peak efficiency.

Use SoC DMA for data transfers whenever available. Instead of using CPU-driven memory copies, configure DMA channels to feed the ping-pong buffer directly from the peripheral or system memory. This method eliminates CPU intervention during data movement, reduces latency between processing list triggers, and frees the host processor for other tasks.

Minimize CPU activity during HSP execution by entering low-power states. Once the processing list is triggered and the DMA is scheduled, the CPU can safely enter sleep mode until a completion flag or interrupt signals at the end of the sequence. This approach significantly reduces dynamic power consumption because the HSP runs autonomously.

4.2 CNN acceleration using the HSP

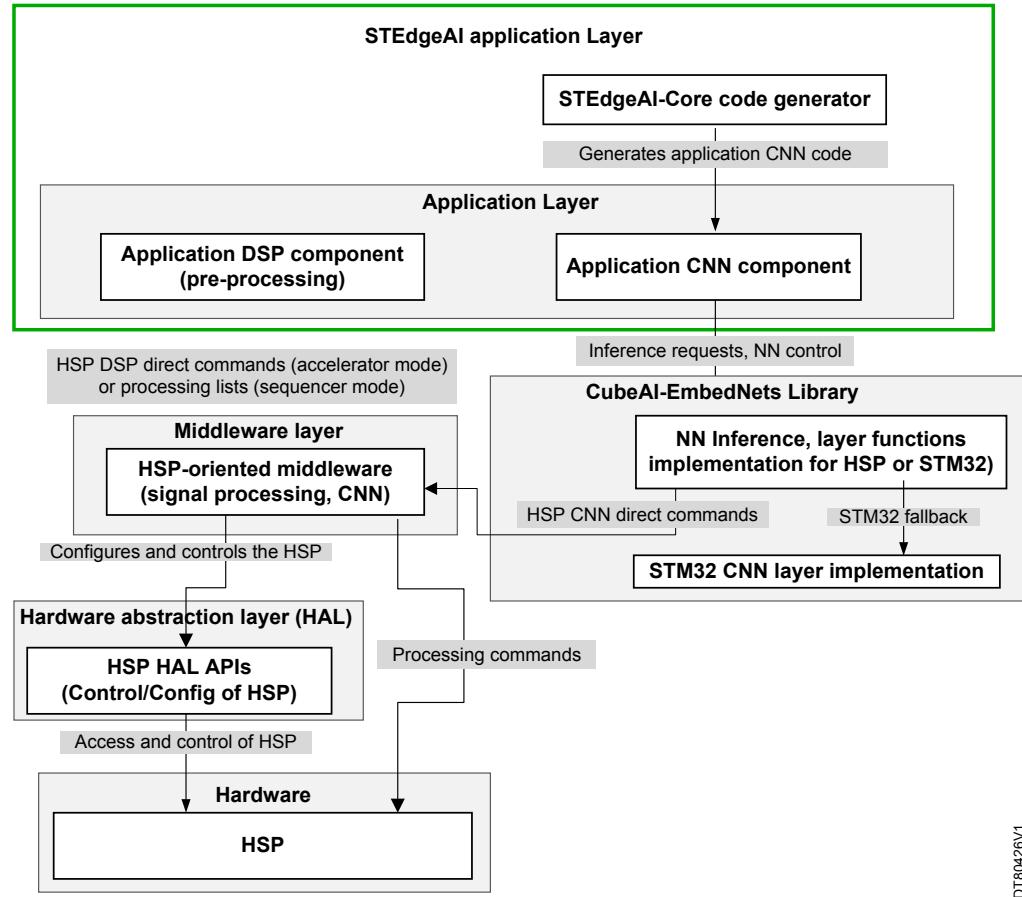
4.2.1 General considerations

Individual layers of a convolutional neural network (CNN) model can be accelerated by using HSP middleware CNN direct commands. Only 8-bit quantized point wise (PW), depth wise (DW), dense (fully connected), Conv2D, and 2D pooling (average or max) layers are supported.

Although it is possible to call CNN direct commands from the HSP middleware, the recommended method is to use [STEdgeAI-Core](#) technology. The code generator produces the most efficient code by selecting appropriate functions. It chooses either HSP-optimized functions or an STM32 software implementation when preferable. The code manages all memory allocation and layer connections.

The figure below provides an overview of how each software component connects to the others.

Figure 9. Overview of interconnections between software components



DT80426V1

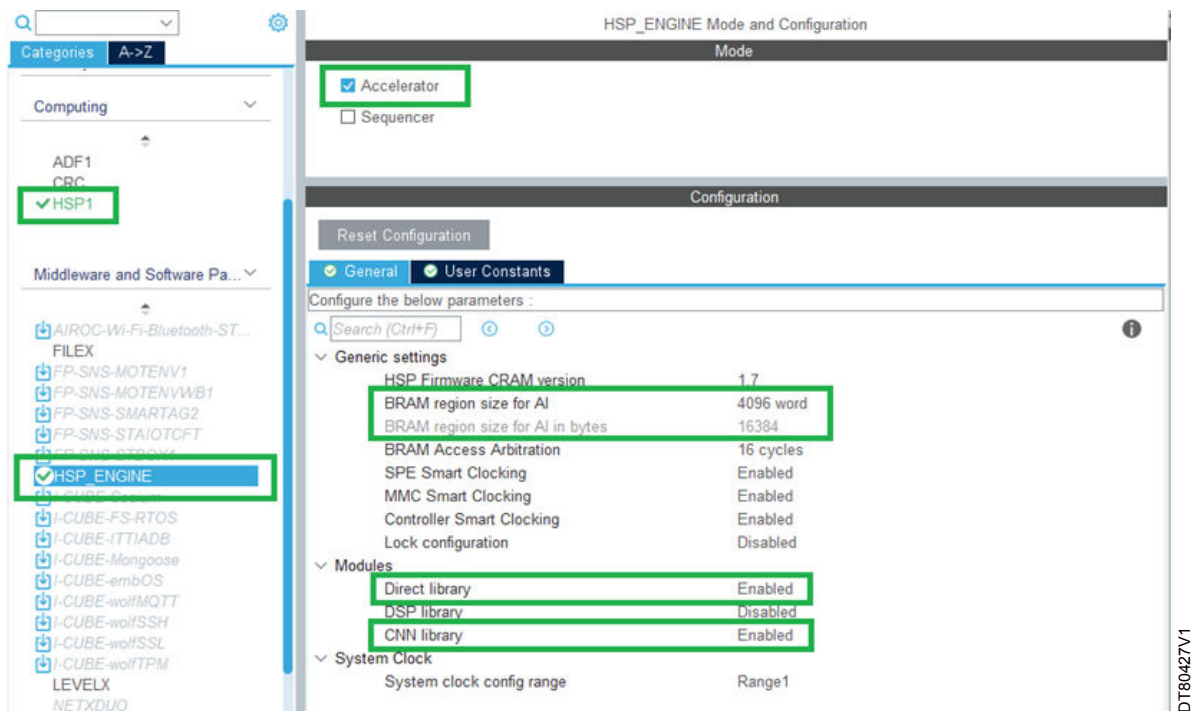
This section focuses on the green box of Figure 9, which explains how to update the application to integrate the execution of 8-bit quantized convolutional neural network (CNN) model inferences by leveraging the acceleration capabilities provided by the HSP. The following section explains how this can coexist with a DSP preprocessing block that also relies on the HSP hardware.

4.2.2 Set up an HSP-enabled project using STM32CubeMX

STM32CubeMX version 6.17 is required.

1. Enable HSP1 in the computing hardware list.
2. Open the HSP_ENGINE panel in Middleware and Software Packs.
3. Select accelerator mode.
4. Enable CNN and direct libraries.
5. Set BRAM region for AI to 16 KB (4096 in 32-bit words). This setting makes the entire BRAM space available for CNN commands.

Figure 10. Setting up an HSP-enabled project using STM32CubeMX



The generated project contains all necessary software components to support DSP and CNN direct operations. These components comprise HSP HAL files, preconfigured configuration files located in the HSP_Engine/target directory, and HSP middleware, which includes DSP and CNN module libraries.

One important parameter to understand is HSP_BRAM_AI_SIZE (configured in STM32CubeMX), which is located in the configuration file hsp_conf.h. This parameter, defined in 32-bit words, specifies the size of an area in BRAM that is usable for AI. This area cannot store any persistent data because AI operations always overwrite it. The remaining BRAM can be used by the DSP application to store constants and memories that AI direct commands do not overwrite. This area functions as overlay memory, addressed in dual-port BRAM-AB access mode by DSP commands and in independent BRAM-A or BRAM-B access mode by AI direct commands.

The global variable hsp_engine_context_t hmw declared in the hsp_engine.c file is the hsp_engine context handler. This variable maintains the state of the HSP and contains all information related to BRAM management. You must keep this symbol name if you do not use STM32CubeMX-generated code. The EmbedNets library requires this external symbol. Using any other name causes a link error.

4.2.3 Add a model to the project using STEdgeAI-Core

Install STEdgeAI-Core version 4.0 or later. The package includes comprehensive documentation. This section focuses on elements relevant to STM32U3B5/3C5 line HSP.

Use main command-line tool stedgeai.exe with the following arguments:

```
stedgeai.exe generate -m my_model.tflite -O time --target stm32u3 -hsp 4096
```

This process produces the following files: network.c/.h, network_data.c/.h and network_details.h

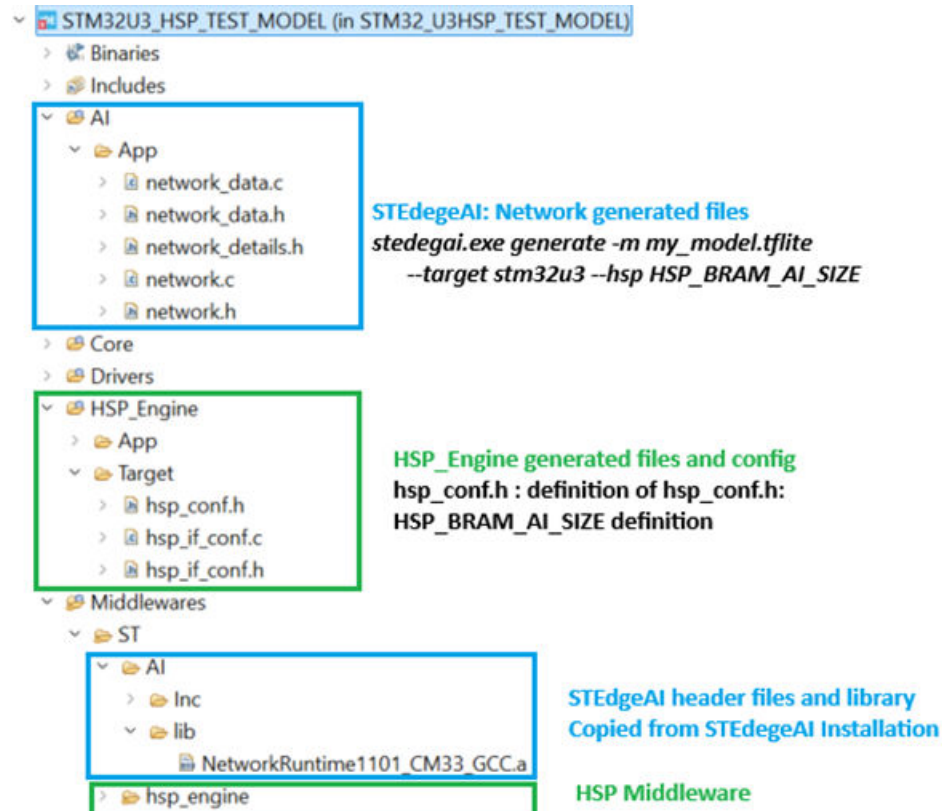
These files contain all elements required to run an inference. The files include the sequence of calls to layer functions from the EmbedNets library, as well as all weights and parameter values.

- The -O time option produces generated code optimized for speed, while the -O ram option provides a solution optimized for memory space. If no option is specified, the code generator defaults to -O balance, which is a compromise between the two previous profiles.
- The --target stm32u3 --hsp 4096 option informs the code generator to create a solution for STM32U3xx using the HSP hardware with a BRAM size of 16 KB (4 × 4096). This ensures that the entire BRAM space is available for AI. The generated code will use HSP enabled layer functions from the EmbedNets Library.

Add the generated files to your project. Update the build environment with the path to the `st_ai` definitions, which can be copied from `Middleware/ST/AI/inc`. Add the EmbedNets library from `Middleware/ST/AI/lib/GCC/ARMCortexM33` if GCC is used. Otherwise, copy the library compiled for IAR Embedded Workbench or KEIL. The following pre-processor definition, `HAVE_NETWORK_INFO` is required to retrieve network information.

The project must have the structure shown in .

Figure 11. Structure of the project using STEdgeAI-Core



DT80428V1

Application initialization and execution

Add the `AI_Init` and `AI_Run` functions to your application. Include the `st_ai.h` and `network.h` header files.

```

/* Global byte buffer to save instantiated C-model network context */
STAI_NETWORK_CONTEXT_DECLARE(network, STAI_NETWORK_CONTEXT_SIZE) stai_network_info info;
/* Global c-array to handle the activations buffer */
STAI_ALIGNED(STAI_NETWORK_ACTIVATION_1_ALIGNMENT)
    static uint8_t activations[STAI_NETWORK_ACTIVATION_1_SIZE_BYTES];

stai_return_code AI_Init()
{
    stai_return_code ret_code;
    ret_code = stai_runtime_init();           // Initialize runtime library
    if (ret_code != STAI_SUCCESS) return ret_code;
    ret_code = stai_network_init(network);     // Initialize network model context
    if (ret_code != STAI_SUCCESS) return ret_code;
    ret_code = stai_network_get_info(network, &info); // Retrieve Network Info
    if (ret_code != STAI_SUCCESS) return ret_code;
    /* Set network activations buffers */
    const stai_ptr acts[] = { activations };
    ret_code = stai_network_set_activations(network, acts, STAI_NETWORK_ACTIVATIONS_NUM);
    if (ret_code != STAI_SUCCESS) return ret_code;
    return STAI_SUCCESS;
}

stai_return_code AI_Run(int8_t *pIn, int8_t *pOut)
{
    stai_return_code ret_code;
    const stai_ptr inputs_ptr[] = { (uint8_t *) pIn };
    const stai_ptr outputs_ptr[] = { (uint8_t *) pOut };
    ret_code = stai_network_set_inputs(network, inputs_ptr, STAI_NETWORK_IN_NUM);
    if (ret_code != STAI_SUCCESS) return ret_code;
    ret_code = stai_network_set_outputs(network, outputs_ptr, STAI_NETWORK_OUT_NUM);
    if (ret_code != STAI_SUCCESS) return ret_code;
    ret_code = stai_network_run(network, STAI_MODE_SYNC);
    if (ret_code != STAI_SUCCESS) return ret_code;
    return STAI_SUCCESS;
}

```

The application code must initialize the HSP_Engine and AI model. It must perform inference calls as required.

```

/* Initialize HSP_ENGINE */
MX_HSP_Engine_Init();

/* AI Initialize */
AI_Init();
while (1)
{
    /* USER CODE END WHILE */
    if (Task_AI_flag)
    {
        Task_AI_flag = 0;
        AI_Process(.....);
    }
}

```

In this example, the `Task_AI_Flag` flag is raised when a DMA transfer completes or when another event, such as new data acquisition, occurs. `pInput` and `pOutput` point to input data and the CNN inference results, respectively.

Inference time can be profiled by collecting cycle counter values before and after calling `AI_Run`.

When the option `--hsp` is not specified to the code generator, the HSP is not used.

Linker file update

STM32U3C5 includes memory sections SRAM1 (192 KB), SRAM2 (64 KB), and SRAM3 (320 KB), which are continuously addressable. However, [STM32CubeMX](#) generates a linker file, *STM32U3C5ZITXQ_FLASH.ld* for [STM32CubeIDE](#) that declares these areas separately. This structure makes it difficult to allocate an activation buffer that does not fit within any single section. Define a single RAM region of 576 KB that covers the entire continuous RAM area.

```

/* Memories definition */
MEMORY
{
    RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 576K
    HSP_DATA_BRAM (rw) : ORIGIN = 0x200A0000, LENGTH = 16K
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 2048K
}

```

4.3 Performance examples for a single model: tinyML[®] benchmarks

ST has published tinyML[®] inference performance results on the MLPerf[®] Inference Benchmark Suite version 1.3 (MLPerf Tiny v1.3), provided by MLCommons Association.

Note: For more information, see the [MLPerf[®] Tiny benchmark page](#).

This section presents inference time measurements comparing STM32U385RG (without HSP) and STM32U3C5ZI (with HSP) for three models available from TinyML as TensorFlow Lite files:

- Image classification (IC): ResNet (*pretrainedResnet_quant.tflite*)
- Keyword spotting (KWS): DSCNN (*kws_ref_model.tflite*)
- Visual wake word (VWW): MobileNet V1 (*vww_96_int8.tflite*)

Note: TensorFlow Lite files are available on [Tiny benchmark training page](#).

Table 3. Comparison of inference duration on STM32U385RG with HSP and STM32U3C5ZI without HSP for three TinyML models

Inference duration	IC	KWS	VWW
STM32U385RG (no HSP)	182.3 ms	48.3 ms	98.64 ms
STM32U3C5ZI (HSP)	49.7 ms	16.54 ms	37.17 ms
Acceleration factor	x 3.67	x 2.92	x 2.65

Figure 12. Inference duration with and without HSP at 96 MHz for three TinyML models

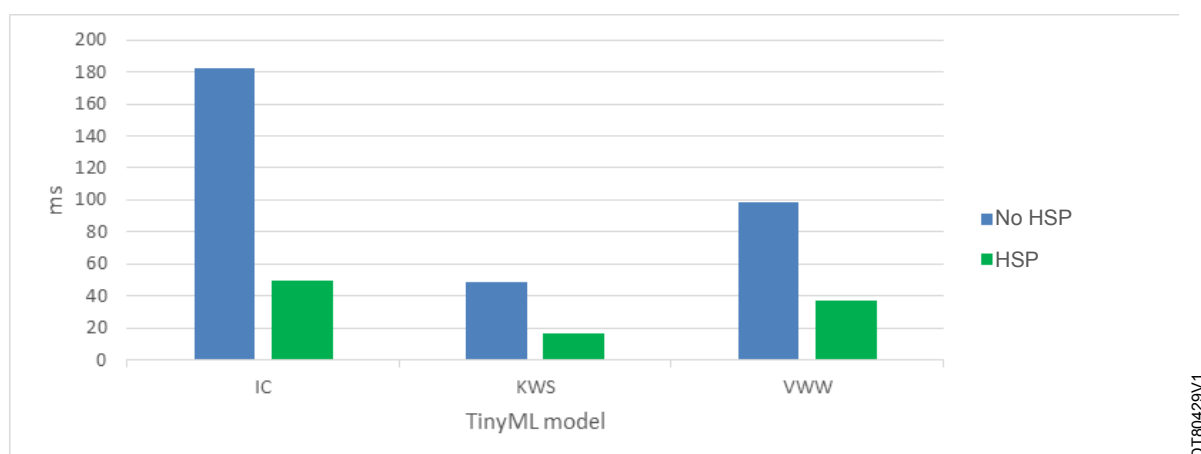
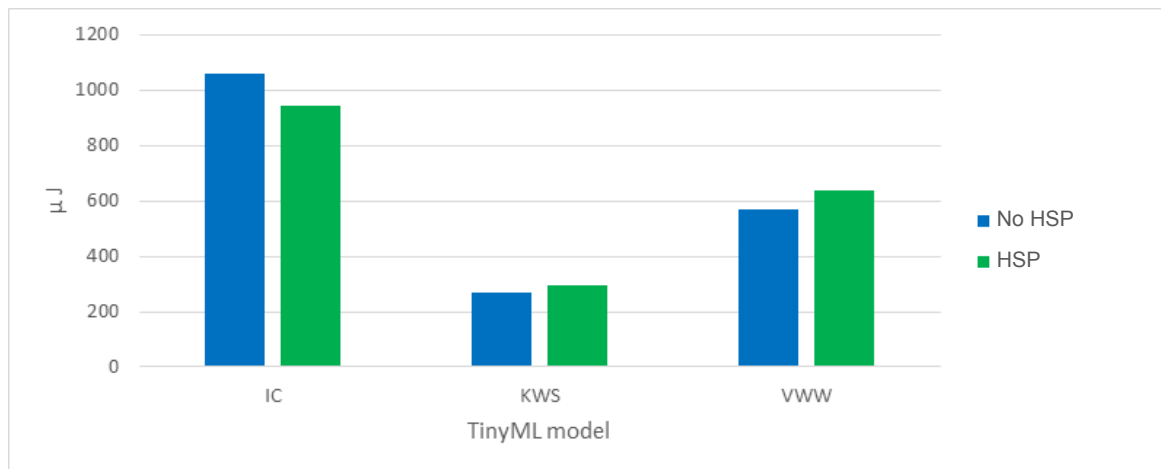


Table 4. Per-inference energy consumption for three TinyML models

Energy per inference	IC	KWS	VWW
STM32U385RG (no HSP)	1062 μ J	271 μ J	568 μ J
STM32U3C5ZI (HSP)	943 μ J	296 μ J	637 μ J
Energy gain	11.21%	9.23%	12.15%

Figure 13. Per-inference energy consumption for three TinyML models



DT80430V1

4.4 Combining DSP and CNN acceleration: streaming wake word

4.4.1 Use case introduction

This section presents the implementation of the MLCommons TinyML Streaming Wake Word benchmark.

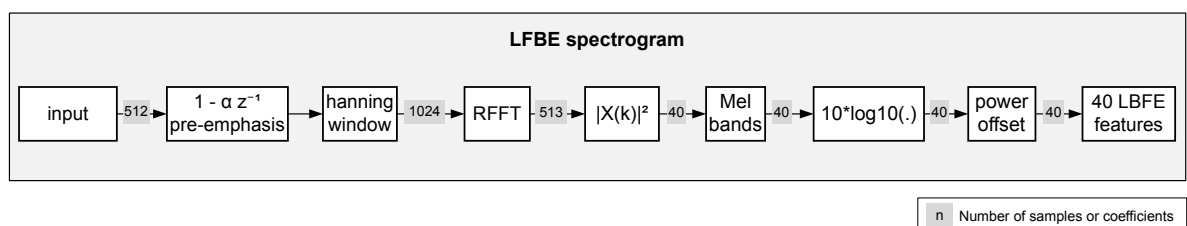
The use case is the detection of an audio keyword, specifically Marvin from the Google Speech Commands dataset.

The audio acquisition layer produces 16 kHz pulse code modulation (PCM) samples. When 512 new samples are available, 40 log frequency band energy (LFBE) features are extracted from the most recent 1024 buffered audio samples, using a stride of 512 samples. The last 30 sets of LFBE features are collected to produce an LFBE spectrogram, which represents approximately one second of audio and has a size of 30×40 . This spectrogram is provided as input to a CNN. The inference result indicates the probability of occurrence for the keyword Marvin. To prevent excessive fluctuations in detection, a simple hysteresis is applied. The final detection result can be sent to the UART output, or the LED can be toggled.

4.4.2 Audio preprocessing: LFBE extraction

The audio pre-processing block to extract 40 log frequency band energies (LFBEs) is described in Figure 14.

Figure 14. Audio pre-processing for LFBE extraction



DT80431V1

Impact on HSP BRAM allocation

Parameters and constants: 6336 bytes

- Hanning window: Only half of the 32-bit floating-point coefficients are stored because of symmetry. This requires 2048 bytes (512 coefficients $\times$ 4 bytes each).
- Bark band coefficients: 987 floating-point values require 3948 bytes.
- Start and end band indices: 40 start indices and 40 end indices, stored as 32-bit integer values, require 320 bytes ($2 \times 40 \times 4$ bytes).
- Other constants: Finite impulse response coefficients (2) and saturation values require 20 bytes.

Static memory: 32 bytes

- Pre-emphasis state buffer.

The DSP audio pre-processing requires a total of 6368 bytes, or 1592 32-bit words, in persistent block RAM (BRAM).

Signal space allocation in nonpersistent BRAM: 4096 bytes

The total BRAM allocation is 10464 bytes, or 2616 32-bit words. This value is less than the available BRAM capacity of 16 KB, or 4096 32-bit words.

Initialization and BRAM allocation

The following source code details the initialization and BRAM allocation for the LBFE feature extraction.

```
void Signal_lfbe_hsp_init(hsp_engine_context_t *p_hmw, lfbeTypeDef_Hsp *p_lfbeStatesHsp, int32_t i32_zero_point,
                        FLOAT f_scale_factor)
{
    p_lfbeStatesHsp->pf_Window = (float32_t *) HSP_BRAM_Malloc(p_hmw, FFT_LENGTH/2, HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pf_MelCoefs = (float32_t *) HSP_BRAM_Malloc(p_hmw, N_MELS_COEFS,
                                                                HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pu32_Melbands_start = (uint32_t *) HSP_BRAM_Malloc(p_hmw, N_MELS,
                                                                HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pu32_Melbands_lens = (uint32_t *) HSP_BRAM_Malloc(p_hmw, N_MELS,
                                                                HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->stateBuf = HSP_BRAM_MallocStateBuffer_Fir(p_hmw, 2, FFT_LENGTH/2,
                                                                HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pf_fir_coef = (float32_t *) HSP_BRAM_Malloc(p_hmw, 2, HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pf_Scratch_audio = (float32_t *) HSP_BRAM_Malloc(p_hmw, FFT_LENGTH/2,
                                                                HSP_BRAM_ALLOCATION_PERSISTENT);
    p_lfbeStatesHsp->pi_satInd = (uint32_t *) HSP_BRAM_Malloc(p_hmw, 1, HSP_BRAM_ALLOCATION_PERSISTENT);

    p_lfbeStatesHsp->i32_frame_cnt = -1;
    p_lfbeStatesHsp->f_preemph_mem = 0;

    p_lfbeStatesHsp->pf_Scratch_fft = (float32_t *) HSP_BRAM_Malloc(p_hmw, FFT_LENGTH, HSP_BRAM_ALLOCATION_DEFAULT);

    p_lfbeStatesHsp->pf_sat[0] = 0.0;
    p_lfbeStatesHsp->pf_sat[1] = 1.0;

    memset(p_lfbeStatesHsp->pf_Scratch_audio, 0, 4 * FFT_LENGTH/2);
    MEM_COPY((int8_t *) p_lfbeStatesHsp->pf_Window, (int8_t *) af_Hamming1024, 4 * FFT_LENGTH/2);
    MEM_COPY((int8_t *) p_lfbeStatesHsp->pf_MelCoefs, (int8_t *) af_Melband_Coefs, 4 * N_MELS_COEFS);
    MEM_COPY((int8_t *) p_lfbeStatesHsp->pu32_Melbands_start, (int8_t *) ai32_Melbands_start, 4 * N_MELS);
    MEM_COPY((int8_t *) p_lfbeStatesHsp->pu32_Melbands_lens, (int8_t *) ai32_Melbands_lens, 4 * N_MELS);

    p_lfbeStatesHsp->f_zero_point = (FLOAT) i32_zero_point;
    p_lfbeStatesHsp->f_scale_factor = 1 / f_scale_factor;

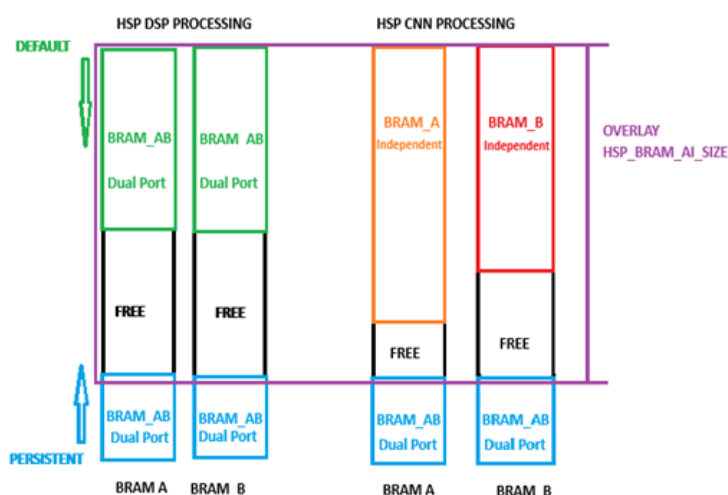
    p_lfbeStatesHsp->f_offset_lfbe = (FLOAT) 0.34214063177502935;
    p_lfbeStatesHsp->f_scale_lfbe = (FLOAT) 0.06785851279738310;

    p_lfbeStatesHsp->pf_fir_coef[0] = 1;
    p_lfbeStatesHsp->pf_fir_coef[1] = -PRE_EMPHASIS_COEF;
}
```

The list below shows the different BRAM addressing schemes based on the types of buffers allocated in BRAM:

- Allocate all constants by using `HSP_BRAM_Malloc` in `HSP_BRAM_ALLOCATION_PERSISTENT` mode: **BRAM_AB** addressing.
- Allocate state buffers (filter memory) by using `HSP_BRAM_MallocStateBuffer_Fir` in `HSP_BRAM_ALLOCATION_PERSISTENT` mode: **dual-port BRAM_AB** addressing.
- Allocate regular buffers with no persistent data by using `HSP_BRAM_Malloc` in `HSP_BRAM_ALLOCATION_DEFAULT` mode: **independent BRAM_A or BRAM_B** addressing.

Figure 15. Buffers allocation in BRAM



DT80432v1

```

/* Initialize HSP_ENGINE */
MX_HSP_Engine_Init();

/* HSP DSP Initialize */
DSP_Preprocess_Init(.....);

/* AI Initialize */
AI_Init();
while (1)
{
    if (Task_AI_flag)
    {
        Task_AI_flag = 0;
        /* HSP DSP PROCESSING : All Dual Port Access (PERSISTENT / DEFAULT) */
        DSP_Preprocess(.....);
        /* HSP CNN PROCESSING : Independant Access CNN allocation BRAM A/B */
        AI_Process(.....);
    }
}

```

Allocating nonpersistent buffers in persistent memory reduces the available space for AI and is not optimal. This allocation can cause more CNN layers to fall back to STM32.

Persistent memory space is always addressed in dual-port mode BRAM_AB, while default memory space can be addressed in dual-port mode.

Main audio preprocessing

The HSP performs the main audio preprocessing in accelerator mode by executing a sequence of direct commands:

```
void Signal_ComputeLfbe_hsp(hsp_engine_context_t *p_hmw,
                           lfbeTypeDef_HSP *p_lfbeStatesHsp,
                           int16_t *pi16_input_audio,
                           int8_t *pi8_output_lfbe)
{
    /* Copy 16bit samples into BRAM */
    MEM_COPY((int8_t *)p_lfbeStatesHsp->pf_Scratch_fft + FFT_LENGTH/2, (int8_t *)pi16_input_audio, FFT_LENGTH);

    /* Convert Qint0 Float */
    HSP_ACC_VectQ16Qf32(p_hmw, (int32_t *)p_lfbeStatesHsp->pf_Scratch_fft + FFT_LENGTH/2,
                       p_lfbeStatesHsp->pf_Scratch_fft + FFT_LENGTH/2, FFT_LENGTH/2);

    /* 2-tap FIR Pre-emphasis */
    HSP_ACC_Fir_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft, p_lfbeStatesHsp->pf_fir_coef,
                  p_lfbeStatesHsp->stateBuf, p_lfbeStatesHsp->pf_Scratch_fft + FFT_LENGTH/2, FFT_LENGTH/2);

    /* Build full frame collecting past samples */
    MEM_COPY((int8_t *)p_lfbeStatesHsp->pf_Scratch_fft,
              (int8_t *)p_lfbeStatesHsp->pf_Scratch_audio, FFT_LENGTH*2);
    MEM_COPY((int8_t *)p_lfbeStatesHsp->pf_Scratch_audio,
              (int8_t *)p_lfbeStatesHsp->pf_Scratch_fft + FFT_LENGTH/2, FFT_LENGTH*2);

    /* Apply Hanning window */
    HSP_ACC_WinSym_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                     p_lfbeStatesHsp->pf_window,
                     p_lfbeStatesHsp->pf_Scratch_fft, FFT_LENGTH, 0);

    /* Perform Real FFT */
    HSP_ACC_Rfft_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft, N_LOG_FFT, 0, 1, 1);

    /* Compute Band Energy (square magnitude) */
    HSP_ACC_CmplxMagSquared_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                              p_lfbeStatesHsp->pf_Scratch_fft, FFT_LENGTH/2 + 1);

    /* Compute Mel Bands */
    HSP_ACC_Fltbank_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                      p_lfbeStatesHsp->pu32_Melbands_start,
                      p_lfbeStatesHsp->pf_MelCoefs,
                      p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);

    /* Compute Log Band Energy */
    HSP_ACC_VectLn_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                     p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);
    /* Get 10*log10(.) and apply Power_Offset, apply clipping */
    HSP_ACC_VectScale_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                        p_lfbeStatesHsp->f_scale_lfbe,
                        p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);
    HSP_ACC_VectOffset_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                         p_lfbeStatesHsp->f_offset_lfbe,
                         p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);
    HSP_ACC_VectSat_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                      p_lfbeStatesHsp->pf_sat,
                      p_lfbeStatesHsp->pf_Scratch_fft,
                      p_lfbeStatesHsp->pi_satInd);

    /* Output formatting (scale/offset), quantize quantization to copy 8bit features to MCU RAM */
    HSP_ACC_VectScale_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                        p_lfbeStatesHsp->f_scale_factor,
                        p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);
    HSP_ACC_VectOffset_f32(p_hmw, p_lfbeStatesHsp->pf_Scratch_fft,
                         p_lfbeStatesHsp->f_zero_point,
                         p_lfbeStatesHsp->pf_Scratch_fft, N_MELS);
    for (int32_t i32_i = 0; i32_i < N_MELS; i32_i++)
    {
        pi8_output_lfbe[i32_i] = __SSAT((int32_t)roundf(p_lfbeStatesHsp->pf_Scratch_fft[i32_i]), 8);
    }
}
```

Output formatting parameters (`p_lfbeStatesHsp->f_scale_factor`, `p_lfbeStatesHsp->f_zero_point`) are the input formatting parameter of the model and can be retrieved from the network info structure.

```
stai_network_info info;
info_stai_network_get_info(network, &info);
p_lfbeStatesHsp->f_scale_factor = info.inputs[0].scale.data[0];
p_lfbeStatesHsp->f_zero_point = info.inputs[0].zeropoint.data[0];
```

HSP performances for LFBE processing

Table 5. HSP performances for LFBE processing

Operation	Arm® Cortex®-M33		HSP		Acceleration ratio
Copy from MCU to BRAM	-	-	670 c ⁽¹⁾	3.46%	-
Q15 ⁽²⁾ to F32 ⁽³⁾	-	-	825 c	4.26%	-
Pre-emphasis (FIR)	-	-	3656 c	18.87%	-
Total pre-emphasis	8235 c	8.60%	5151 c	26.58%	x 1.60
Signal copy	-	-	2471 c	12.75%	-
Windowing	-	-	1625 c	8.39%	-
Total windowing	6679 c	6.98%	4096 c	21.14%	x 1.63
Real FFT	59748 c	62.40%	4850 c	25.03%	x 12.32
Squared magnitude	5672 c	5.92%	1083 c	5.59%	x 5.24
Mel bands	-	-	1710 c	8.82%	-
Log operation	-	-	1298 c	6.70%	-
Power scalin/offset	-	-	349 c	1.80%	-
Total log Mel bands	14518 c	15.16%	3357 c	17.32%	x 4.32
Output formatting	-	-	198 c	1.02%	-
Copy from BRAM to MCU	-	-	644 c	3.32%	-
Total output formatting	898 c	0.94%	842 c	4.34%	x 1.07
Total cycles	95750 c	-	19379 c	-	x 4.94
Total duration ⁽⁴⁾	0.997 ms	-	0.202 ms	-	-

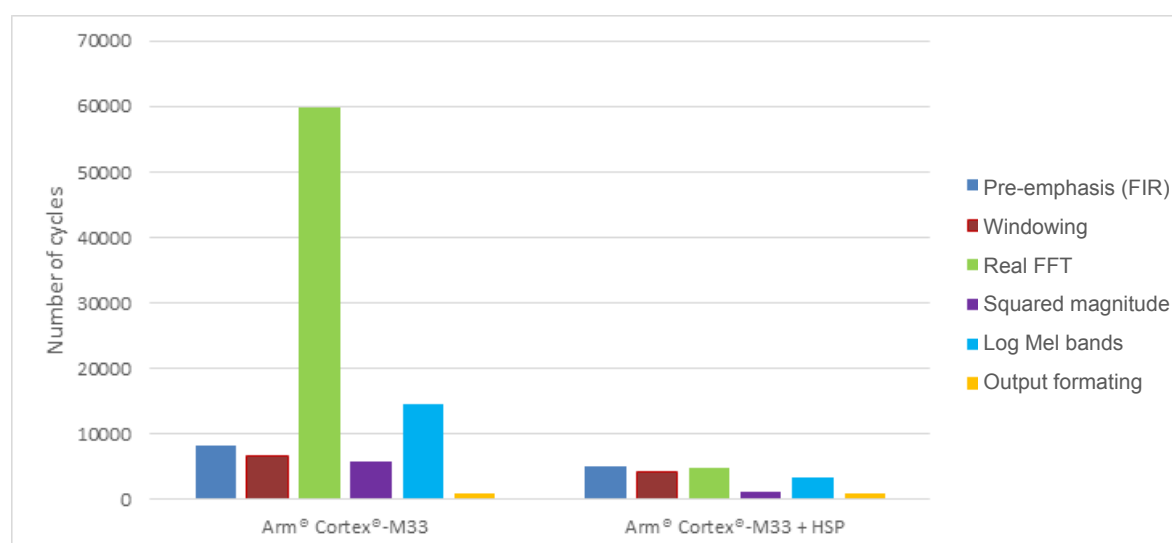
1. "c" stands for cycles

2. Q15 format represents a fixed-point number with 1 sign bit and 15 fractional bits.

3. F32 stands for 32-bit floating point

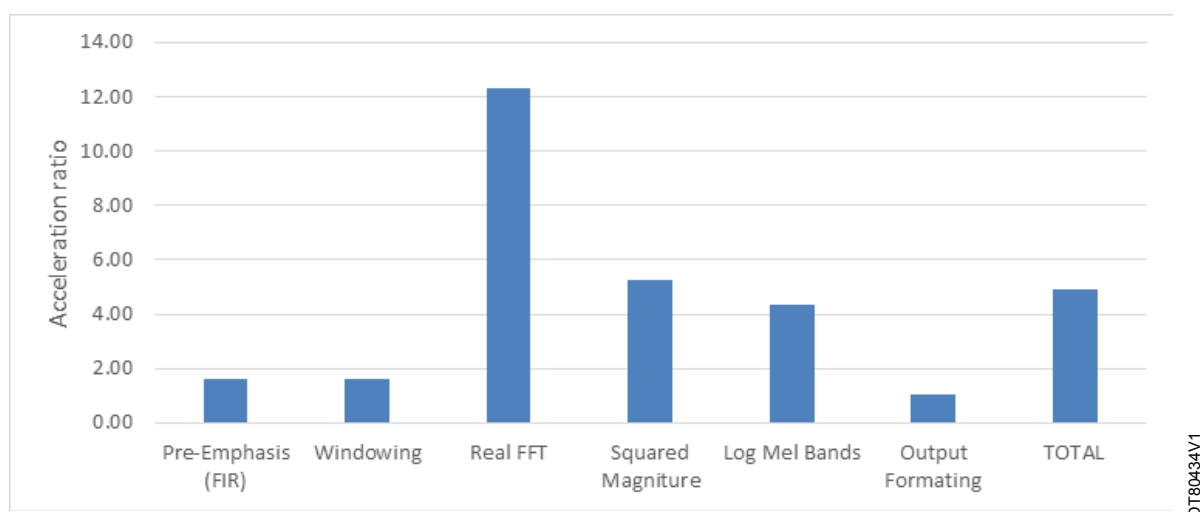
4. The total duration is measured at a running frequency of 96 MHz.

Figure 16. DSP preprocessing steps in cycles



DT80433V1

Figure 17. Acceleration of HSP compared with Arm® Cortex®-M33



DT80434V1

The HSP acceleration ratio can reach 12 for a single real fast Fourier transform (FFT) but the average speed factor is limited to 5:

- Some operators are only modestly accelerated. For example, a small 2-tap FIR filter.
- Functions with a low ratio of processing to memory transfer show limited improvement.
- HSP processing requires additional memory transfers between the MCU and HSP_BRAM.

4.4.3 CNN performances and BRAM management

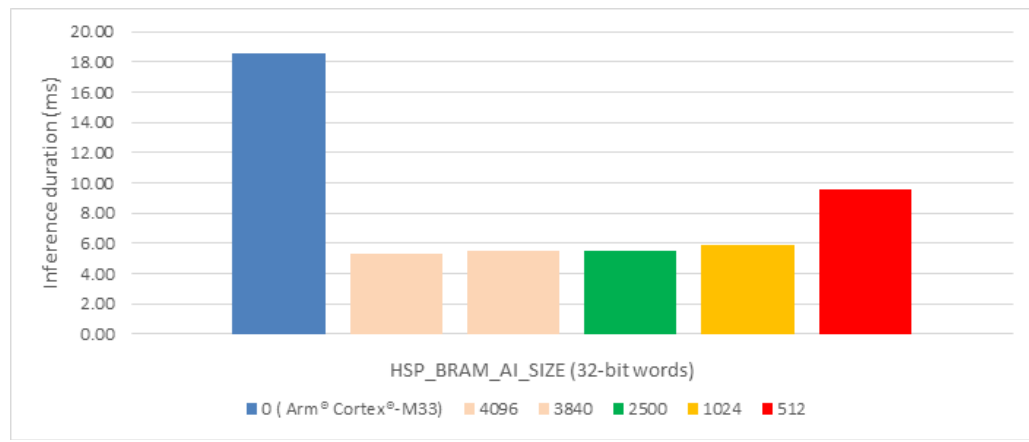
The model is a compact version of MatchBoxNet, which is a one-dimensional depth wise separable convolutional neural network (1D-DSCNN).

A TensorFlow Lite (TFLite) model file is available from the MLCommons TinyML git repository. The file name is *str_ww_ref_model.tflite* (see the [MLPerf® Tiny benchmark page](#)).

Note: In this section, "words" refers to 32-bit data.

Table 6. Inference performance as a function of HSP_BRAM_AI_SIZE

Measure	HSP_BRAM_AI_SIZE (32-bit words)					
	0	4096	3840	2500	1024	512
Inference duration	18.53 ms	5.34 ms	5.52 ms	5.52 ms	5.86 ms	9.61 ms
HSP acceleration	-	x 3.47	x 3.36	x 3.36	x 3.16	x 1.93
Acceleration loss	-	-	3.2%	3.2%	8.8%	44.4%

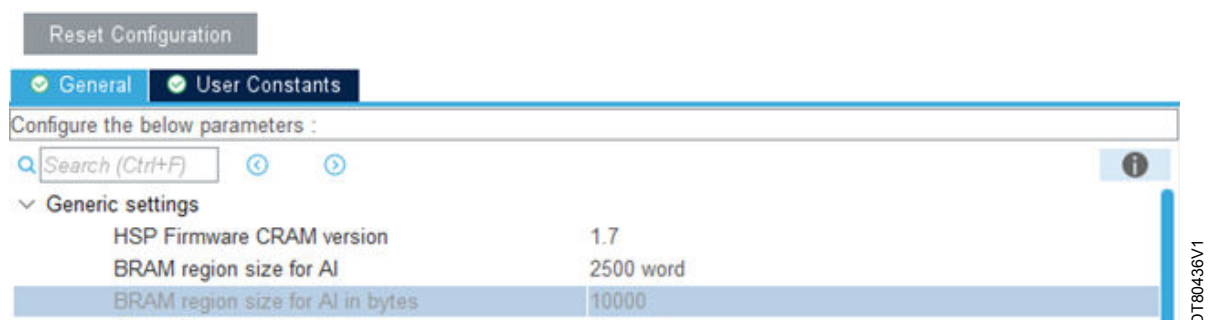
Figure 18. Inference duration as a function of HSP_BRAM_AI_SIZE in 32 bit words


Audio preprocessing requires approximately 1592 words of persistent space in HSP_BRAM, leaving 2504 words available for AI processing. With 2500 words available, performance decreases by only 3.22 percent, which corresponds to an additional inference time of 0.22 milliseconds. This impact is minimal. Performance degradation becomes more significant due to increased fallback to the Arm® Cortex®-M33 core when HSP_BRAM_AI_SIZE is set to 1024 words (4 KB).

Set HSP_BRAM_AI_SIZE to 2500 in the *HSP_Engine/Target/hsp_conf.h* file.

```
/* BRAM ----- */
/**
 * @brief Define the BRAM size reserved for CNN processing
 */
#define HSP_BRAM_AI_SIZE 2500U /*!< Size in word */
```

This task can also be performed in **STM32CubeMX** by setting the value in the HSP_Engine middleware panel.

Figure 19. BRAM region size for AI configuration


Network files should therefore be produced using the following arguments:

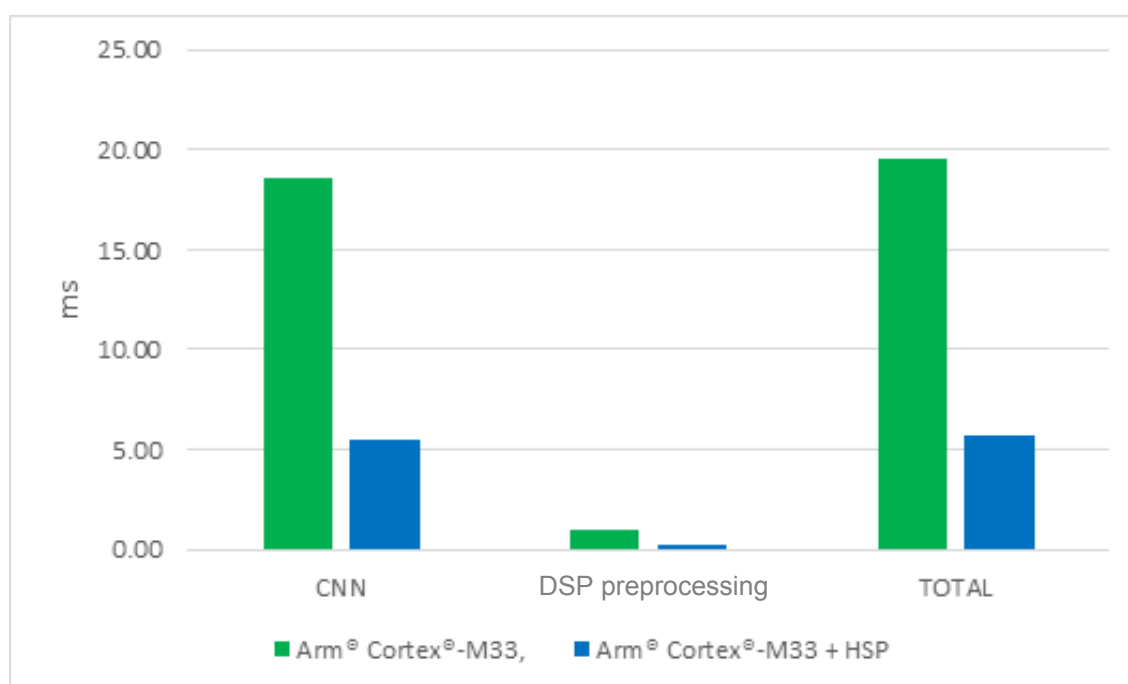
```
stedgeai.exe generate -m str_ww_ref_model.tflite -O time --target stm32u3 -hsp 2500
```

4.4.4 Combined DSP and CNN performances

Table 7. Combined CNN and DSP preprocessing duration with and without HSP

	Arm® Cortex®-M33		Acceleration ratio
	without HSP	with HSP	
CNN	18.53 ms	5.52 ms	x 3.36
DSP preprocessing	1.00 ms	0.20 ms	x 4.94
Total	19.53 ms	5.72 ms	x 3.41

Figure 20. Combined CNN and DSP preprocessing duration with and without HSP



DT80437V1

The duration of CNN inference is approximately 20 times longer than the duration of DSP feature extraction. The optimization decision must prioritize CNN requirements. Always allocate as much HSP_BRAM_AI_SIZE as necessary to achieve CNN performance targets. Allocate the entire BRAM if required.

If no memory space is available for DSP persistent data, the most efficient alternative is to save the data in the MCU RAM and restore it to the BRAM before each DSP processing block.

This approach incurs a performance penalty. However, the penalty is typically less significant than the performance degradation that results from insufficient BRAM for the CNN.

5 Debug scenarios and how to use them

Debugging the hardware signal processor (HSP) is essential to validate real-time behavior, diagnose errors, and optimize performance. The HSP provides hardware-assisted debug features such as break interface (BKITF), timestamp capture (TSC), task comparator unit (TCU), and snoop bus (SNP). These features allow the user to stop execution at precise points, trace the processing flow, and capture timing information.

Before enabling any debug feature, ensure that the HSP is properly initialized:

- Firmware loading: Copy the HSP firmware into CRAM/DRAM and initialize using the FW_INIT command.
- Activation: Set the BOOTEN bit in HSP_CR to 1. This enables the HSP core clock and starts the sequencer.
- Processing lists ready: All PLs must be recorded in CRAM and their parameters stored in DRAM.

Note: The middleware functions perform the operations.

- Memory accessibility: Verify that BRAM-AB buffers are instantiated and accessible via the debugger.

For synchronized debugging, configure the cross-trigger interface (CTI) or use DBGMCU so that when the CPU halts, the HSP halts too, and vice versa. Example: Program CTI channels to propagate break signals between CPU and HSP. This is essential when debugging interactions between an application code and HSP processing lists.

Note: CTI is not available on all products. See the product documentation for availability.

Cross-break ensures both processors stop simultaneously, preserving system state and avoiding race conditions. This is the recommended approach for complex debugging scenarios.

Below are the most common debug scenarios explained in detail, with step-by-step guidance and examples.

5.1 Break at the start of the processing list

This scenario is ideal when the user wants to verify that a specific processing list (PL) starts correctly and inspects its initial state. The HSP uses task comparators to detect when a specific list begins execution. This method does not require to generate a specific processing list for the debug.

How to configure

- Program the task comparator (TCU) with the task number that corresponds to the processing list (PL).
- Enable the comparator and configure BKITF to filter breaks by this task.
- Optionally:
 - Enable a single-step mode so the HSP stops at each processing function (PF) inside the processing list.
 - Freeze the HSPDMA when SPE breaks.

Note: The HSPDMA is not available on all devices, verify product capabilities.

- Store the processing list number and information on break.
- Allow the interrupt when SPE breaks.

Example

Suppose the PL for FFT processing is task number five, and the comparator 0 is used:

```
HSP_TCUCFGR.TSKCMP0[5:0] = 5 // Comparator 0 watches task 5
HSP_TCUCFGR.TCOEN = 1      // Enable comparator 0
HSP_BKICFGR.TSKFLT = 1     // Filter break by task comparator
HSP_BKICFGR.SSEN = 1       // Enable single step (optional)
HSP_HSPCR.HSPDMA_FZ = 1    // Froze HSPDMA on break (optional)
HSP_CAPCR.CAPMOD[1:0] = 1  // Capture PL/PF info on break (optional)
HSP_HSP_ERR_IER.BKINIE = 1 // Generate an interruption on break (optional)
```

What happens

When task 5 starts, the HSP halts. An interrupt is generated if enabled. Read HSP_CAPDR register to view the PL and PF context, inspect BRAM buffers, then resume by writing HSP_ERR_ICR.BKINC = 1. If the single-step mode is enabled, the HSP stops again after each PF call.

Note: To stop doing step-by-step debug, set HSP_BKICFGR.SSEN to 0. Setting HSP_BKICFGR.TSKFLT to 0 while HSP_BKICFGR.SSEN = 1 keeps the step-by-step debug feature outside the processing list.

CPU behavior

If CTI is not configured, the CPU continues to run and can alter shared memory. For synchronized debugging, enable CTI or DBGMCU so that the CPU halts when the HSP halts. This prevents race conditions and ensures consistent state inspection.

Note: CTI is not available on all products. See the product documentation for availability.

5.2 Break before executing a specific processing function

This scenario is ideal when the user needs to check intermediate results before a critical function, for example, before applying a filter or transform. The user can break execution just before the processing function runs. To break before a specific processing function, the user must modify the processing list to insert a HSP_SEQ_SetFlag instruction before the target function. Flags 28 to 31 enable the break interface.

How to configure

To configure the break before the desired processing function in the processing list, follow these steps:

1. Insert SET_FLAG[28:31] before the required processing function in the processing list.
2. Configure the hardware as follows:
 - Disable task-based break by clearing the BKITF bit.
 - Enable break on the selected flag number.
 - Disable single-step mode.
 - Optionally
 - Freeze the HSPDMA when the SPE breaks.

Note: The HSPDMA is not available on all devices, verify product capabilities.

- Store the processing list number and information on break.
- Allow the interrupt when the SPE breaks.

Example

To break before the HSP_SEQ_VectScale_f32() function in the processing list by using flag 28, perform the following steps:

Original processing list	Modified processing list with a breakpoint
<pre> HSP_SEQ_StartRecordPL(PLX); HSP_SEQ_VectAvg_f32(...); HSP_SEQ_IfFloat(IFEQ,...); HSP_SEQ_VectMul_f32(...); HSP_SEQ_IfElse(); HSP_SEQ_VectMax_f32(...); HSP_SEQ_IfFloat(IFGT,...); HSP_SEQ_IfFloat(IFLE,...); HSP_SEQ_VectScale_f32(...); HSP_SEQ_IfElse(); HSP_SEQ_VectOffset_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfElse(); HSP_SEQ_VectScale_f32(...); FIR_F32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfEndif(); HSP_SEQ_IfFloat(IFNE,...); HSP_SEQ_VecSqrt_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfFloat(IFGE,...); HSP_SEQ_VectSat_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_StopRecordPL(); </pre>	<pre> HSP_SEQ_StartRecordPL(PLX); HSP_SEQ_VectAvg_f32(...); HSP_SEQ_IfFloat(IFEQ,...); HSP_SEQ_VectMul_f32(...); HSP_SEQ_IfElse(); HSP_SEQ_VectMax_f32(...); HSP_SEQ_IfFloat(IFGT,...); HSP_SEQ_IfFloat(IFLE,...); HSP_SEQ_IfFloat(IFLE,...); HSP_SEQ_SetFlag(...,28); HSP_SEQ_VectScale_f32(...); HSP_SEQ_IfElse(); HSP_SEQ_VectOffset_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfElse(HSP_SEQ_VectScale_f32(...); FIR_F32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfEndif(); HSP_SEQ_IfFloat(IFNE,...); HSP_SEQ_VecSqrt_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_IfFloat(IFGE,...); HSP_SEQ_VectSat_f32(...); HSP_SEQ_IfEndif(); HSP_SEQ_StopRecordPL(); </pre>

```
HSP_BKICFGR.TSKFLT = 0 // Filter break by task comparator
HSP_BKICFGR.PFCT28EN = 1 // Enable break on flag 28
HSP_BKICFGR.SSEN = 0 // Disable single step mode debug
HSP_HSPCR.HSPDMA_FZ = 1 // Froze HSPDMA on break (optional)
HSP_CAPCR.CAPMOD[1:0] = 1 // Capture PL/PF info on break (optional)
HSP_HSP_ERR_IER.BKINIE = 1 // Generate an interruption on break (optional)
```

What happens

The HSP stops before executing `HSP_SEQ_VectScale_f32()`. If an interruption is enabled, an interruption is generated. The user can inspect buffers and then resume operation. To step through subsequent PFs, enable SSEN and clear all the active break flags before writing BKINC.

Note: *Inline functions do not generate CALL markers, so they cannot trigger breaks.*

CPU behavior

The CPU remains active unless the cross-trigger interface (CTI or DBGMCU) is enabled. If the CPU continues to run, it can trigger new events or modify shared buffers, which complicates debugging. Enable CTI or DBGMCU to synchronize halts.

Note: *CTI is not available on all products. See the product documentation for availability.*

5.3 Break on processing error

It is useful to diagnose faults such as:

- BRAM access violation (ACCERRF)
- FPU computation error (FPUERRF)
- Illegal opcode (OPCOERRF)
- Firmware error (FWERRF)
- FPU saturation (FPUSATF)

How to configure

- Enable error sources in HSP_BKICFGR register.
- Optionally:
 - Enable an interruption generation: `HSP_ERR_IER.BKINIE = 1`.
 - Enable TSC for context capture.

Example

To break on FPU saturation:

```
HSP_BKICFGR.FPUSATEN = 1
HSP_ERR_IER.BKINIE = 1
HSP_CAPCR.CAPMOD[1:0] = 1 // Capture PL/PF info on break (optional)
```

What happens

When an error occurs, the HSP halts. Read HSP_CAPDR for PL/PF context and HSP_ERRINFR for root cause. Clear the error flag first, then write `BKINC = 1` to resume.

CPU behavior

If CTI or DBGMCU is active, the CPU halts immediately, which allows inspection of both states. If CTI or DBGMCU is not active, the CPU continues running, which may overwrite error-related data in shared memory.

Note: *CTI is not available on all products. See the product documentation for availability.*

5.4 Asynchronous break

This feature allows the user to interrupt the HSP at any time during execution. Once it is enabled, the HSP stops at the next processing function.

How to configure

- Enable a single-step mode so that the HSP stops at each processing function inside the PL.
- Optionally:
 - Program and enable a task comparator with the task number corresponding to your PL.
 - Freeze the HSPDMA when SPE breaks.

Note: *The HSPDMA is not available on all devices, verify product capabilities.*

- Store the processing list number and information on break.
- Allow the interrupt when SPE breaks.

Example

If you suspect an issue in a long-running processing list (PL) number 5 and want to inspect the memory content during the next processing function, you can configure a task comparator on the filtered task. Alternatively, use the CAPMOD to store the processing list number and the function that was executed.

```
HSP_BKICFGR.SSEN = 1           // Enable single step mode
HSP_TCUCFGR.TSKCMP0[5:0] = 5   // Comparator 0 watches task 5 (optional)
HSP_TCUCFGR.TC0EN = 1         // Enable comparator 0 (optional)
HSP_HSPCR.HSPDMA_FZ = 1       // Froze HSPDMA on break (optional)
HSP_CAPCR.CAPMOD[1:0] = 1     // At every break store PL number and PF (optional)
HSP_HSP_ERR_IER.BKINIE = 1    // Generate an interruption when the HSP breaks (optional)
```

What happens

The HSP halts at the next PF call. An interruption error is generated if enabled. Inspect the buffers, then resume with BKINC = 1. The HSP stops again after each PF call until you clear SSEN.

CPU behavior

The CPU continues to run unless the CTI is configured. If the CPU modifies shared buffers during the halt, debugging results can be inconsistent.

Note: *Inline functions do not generate CALL markers, so they cannot trigger breaks.*

6 Trace and profiling features on the HSP

The HSP typically executes complex signal processing tasks autonomously, triggered by various events. Interrupting execution with breakpoints affects timing behavior, making it impossible to measure actual performance. Trace and profiling provide the following benefits:

- **Execution visibility:** Identify which processing list or function runs at any given time.
- **Timing analysis:** Determine the duration required to execute each processing function or list.
- **Traffic detection:** Detect functions that consume excessive cycles.

The HSP provides three complementary trace mechanisms described in the following sections.

6.1 Time-stamp capture (TSC) block

The TSC block is a powerful hardware feature that allows capturing execution history without external pins or intrusive breakpoints. It records:

- **Timestamp:** A free-running counter value for precise timing.
- **Task number (TSKNB):** Indicates which processing list is active.
- **Processing function number (PFCTNB):** Identifies the function being executed.
- **Conditional execution history:** Managed through IFCNT and IFHIST fields.

Role of IFCNT and IFHIST

If counter IFCNT

- Incremented every time an inline conditional function (`__IF_ELSE`) is executed.
- Represents the number of conditional evaluations since the last processing function call.

If history IFHIST

- A bitfield that stores the result of each conditional evaluation:
 - Bit set to 1 if the condition was TRUE.
 - Bit remains 0 if FALSE.
- Reset when:
 - A non-inline processing function executes.
 - The end of the processing list is reached.

When analyzing the execution flow, IFCNT and IFHIST help reconstruct the decision path taken during PL execution.

For example, IFCNT = 3 and IFHIST = 0b101 mean that three conditions are evaluated, and the first and third conditions are TRUE.

How to configure TSC for profiling

To enable detailed profiling, follow these steps:

1. On HSP_CAPCR register:
 - If the user needs the timestamp function, then PRESC must be set to the wanted value, and FRCNTEN must be set to 1. In this mode, IFCNT and IFHIST are not logged. This mode can be used to check the performances.
 - If the user wants to trace the processing execution, FRCNTEN must be set to 0. In this mode, IFCNT and IFHIST are logged allowing to trace the processing list execution.
 - Set CAPMOD to 3 to capture the start of execution of every processing function.

2. The captured data can be read in different ways:

- Poll the CAPRDYF flag, then read the HSP_CAPDR register.
- Use an interrupt service by setting the CAPRDYIE bit of the HSP_EVT_IER register. When the interrupt is asserted, read the HSP_CAPDR register.
- Use SoCDMA by selecting the tsc_trgo event in the HSP_TRGOCFGR register. In this case, the TSC generates a trigger pulse every time data is ready in the HSP_CAPDR register. Program SoCDMA to read the HSP_CAPDR register and store the data in a memory buffer.
- Each record includes the following fields:
 - Timestamp
 - TSKNB (task number)
 - PFCTNB (processing function number)
 - IFCNT and IFHIST (conditional history)

Note: The CPU must read this register faster than the recording process to avoid overrun.

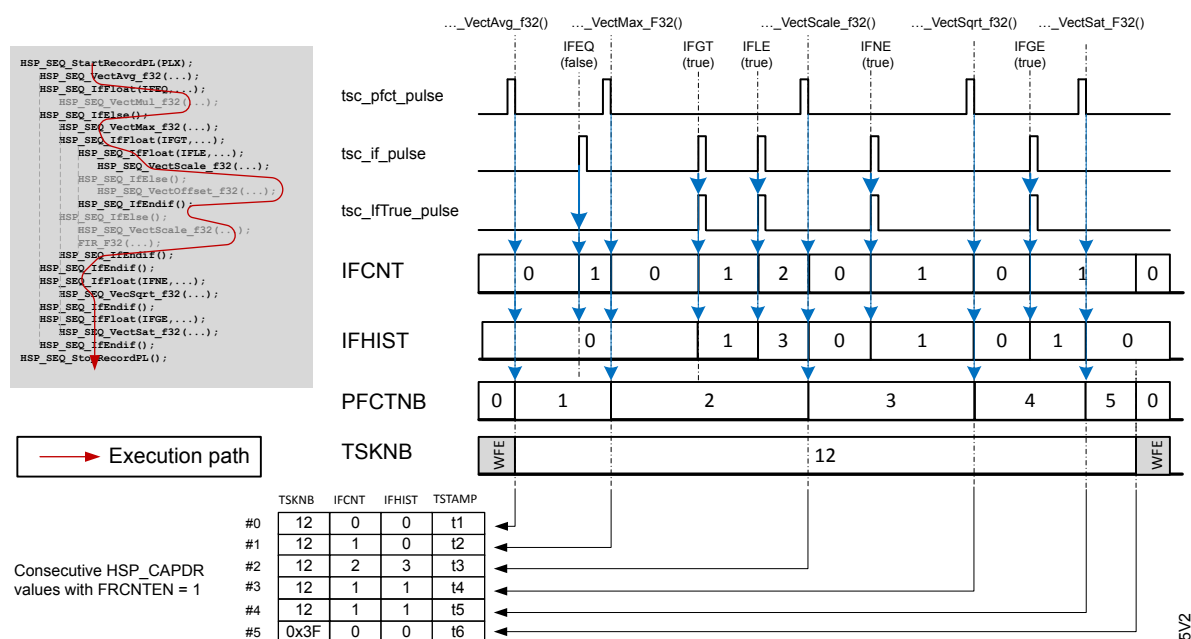
3. Interpret IFCNT and IFHIST:

- Use IFCNT to determine how many inline conditions were processed.
- Decode IFHIST to identify which conditions were true or false.

Example

We aim to analyze the execution flow of the following processing list, PL12:

Figure 21. TSC capture example



Captured CAPDR

Timestamp: t2-t1, TSKNB: 12, IFCNT: 5, IFHIST: 0b11110 (0x1E)

Interpretation

Six records are captured (records #0 to #5 in [Figure 21](#)):

- The first record is captured when the processing list executes the first processing function, HSP_SEQ_VectAvg_f32(). The timestamp of the first function executed and the processing list number are stored.
- The second record is captured when the processing list executes the function HSP_SEQ_VectMax_f32() (see record #1). IFCNT = 1 and IFHIST = 0, which indicates that the processing list executed one __IF and the condition was false. By correlating with the source code, it is possible to deduce that the __ELSE part was executed, and thus the HSP_SEQ_VectMax_f32() function.
- The third record is captured when the processing list executes the function HSP_SEQ_VectScale_f32() (see record #2). IFCNT = 2 and IFHIST = 3, which indicates that the processing list executed two __IF statements. The condition on the first __IF was true (IFHIST[0] = 1). The condition on the second __IF was also true (IFHIST[1] = 1). By correlating with the source code, it is possible to deduce that the HSP_SEQ_VectScale_f32() function was executed.
- The fourth record is captured when the processing list executes the function HSP_SEQ_VectSqrt_f32() (see record #3). IFCNT = 1 and IFHIST = 1, which indicates that the processing list executed one __IF and the condition was true. By correlating with the source code, it is possible to deduce that the HSP_SEQ_VectSqrt_f32() function was executed.
- The fifth record is captured when the processing list executes the function HSP_SEQ_VectSat_f32() (see record #4). IFCNT = 1 and IFHIST = 1, which indicates that the processing list executed one __IF and the condition was true. By correlating with the source code, it is possible to deduce that the HSP_SEQ_VectSat_f32() function was executed.
- The sixth record is captured when the processing list returns to WFE. This is observed because TSKNB = 0x3F (idle task).

The execution path, shown in red in [Figure 21](#), can be reconstructed for debugging or optimization.

Note: *tsc_pfct_pulse, tsc_if_pulse and tsc>IfTrue_pulse signals are available on the SNP interface. This availability allows users to capture a real-time trace of HSP activity.*

6.2 Task comparators

Task comparators monitor the execution of specific processing lists. When the selected processing list execution starts or stops, the comparator output toggles. You can observe the output through the SNP interface. External tools, such as logic analyzers or trace probes, can capture the output.

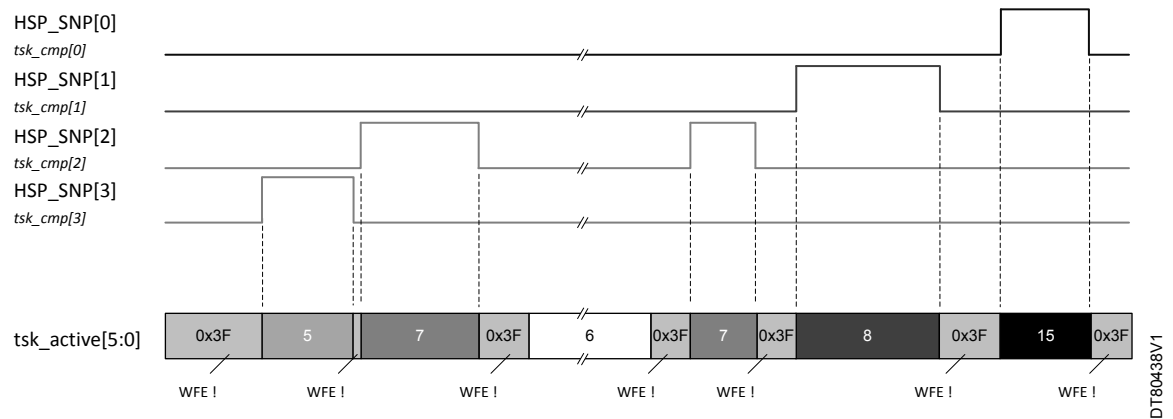
- Up to four comparators are available.
- Each comparator is programmable with a different processing list number.
- Outputs are routable to HSP_SNP pins for external observation.

Note: *Refer to the product datasheets [\[3\]](#) for information about pin availability.*

Example

We want to measure how frequently each of the four processing lists PL15, PL8, PL7, and PL5, is used in a control loop.

Figure 22. Processing list monitoring



How to configure

```
// Task Comparator Programming Example

// Configure comparators for PL15, PL8, PL7, PL5
HSP_TCUCFGR.TSKCMP0 = 15; // Comparator 0 for PL15
HSP_TCUCFGR.TSKCMP1 = 8;  // Comparator 1 for PL8
HSP_TCUCFGR.TSKCMP2 = 7;  // Comparator 2 for PL7
HSP_TCUCFGR.TSKCMP3 = 5;  // Comparator 3 for PL5

// Enable comparators
HSP_TCUCFGR.TC0EN = 1;
HSP_TCUCFGR.TC1EN = 1;
HSP_TCUCFGR.TC2EN = 1;
HSP_TCUCFGR.TC3EN = 1;

// Route SNP outputs to pins
HSP_SNPR.SNPSELA = 5; // Select comparator signals for SNP outputs
HSP_SNPR.SNPSELB = 5;
```

6.3

Tsk_active signals

The TSK_ACTIVE[5:0] signals indicate which task, or processing list, is active. These signals can be routed to the SNP interface for real-time observation.

- Provide a continuous view of scheduling.
- Useful for verifying that event priorities and tasks behave as expected.

How to configure

```
// Route SNP outputs to pins
HSP_SNPR.SNPSELA = 1; // Select tsk_active[5:0] signals for SNP outputs
HSP_SNPR.SNPSELB = 1;
```

7 HSP DSP performance

Most HSP processing functions can be executed in two modes:

- **Accelerator mode:** immediate execution of a single processing function.
- **Sequencer mode:** recording one or more functions into a processing list for execution upon an event.

Direct register programming of the HSP is not required in typical workflows. The middleware functions provide a higher-level interface and are the recommended ways to use HSP capabilities.

The middleware exposes four function families, identifiable by their prefix:

- **HSP_ACC_:** Direct command (accelerator mode). Executes a single HSP function immediately.
- **stm32_hsp_:** CMSIS-DSP compatible wrappers (accelerator mode). Mirrors Arm® CMSIS-DSP function signatures for straightforward code reuse; internally issues a direct command.
- **HSP_ACC_ (CNN):** Direct command for CNN layers (accelerator mode). Compatible with STM32 AI tooling; runs a single layer on the HSP.
- **HSP_SEQ_:** Processing-list recording (sequencer mode). Adds a function to a list that the sequencer executes later.

These middleware functions are wrappers over the same HSP kernel. For example,

`HSP_SEQ_CmplxConj_f32()`, `HSP_ACC_CmplxConj_f32()`, and `stm32_hsp_cmplx_conj_f32()` all execute the same processing function: `CMPLX_CONJ_F32`.

Accelerator mode (HSP_ACC_*, stm32_hsp_*)

Execution is issued immediately as a direct command: the middleware programs HSP registers at call time, the engine decodes the command and runs the kernel. This introduces a small overhead (command programming/decoding and status readback). In practice, the overhead is generally negligible for vector operations and depends on factors such as number of parameters, system clock configuration, and bus traffic.

Sequencer mode (HSP_SEQ_*)

Execution is recorded during the processing list (PL) creation phase. Programming and decoding occur at recording time, not at runtime. When an event triggers the PL, the HSP jumps to compiled code. It runs the kernel without direct command overhead during execution. This action achieves the raw kernel performance of the function.

The tables listed here show the runtime execution duration that is measured for each function executed by the sequencer. For example, the figure associated with `HSP_SEQ_CmplxConj_f32()` is the execution time of the conjugate function at runtime; it does not include the cost to record that operation in a PL. Recording overhead is paid once during PL creation and is not part of the runtime measurement.

The performances of the processing functions are generally predictable, but subject to variation due to BRAM-AB activity. The table below gives the performances when there is no conflict in the BRAM-AB (BARB = 0). See [Section 3.4](#) for more details about BARB.

Note: The values provided in [Table 8](#) are measured at the same running frequency.
Only some functions are listed in [Table 8](#). For the complete list, see the user manual [\[2\]](#).

Table 8. Performance of middleware functions

HSP middleware functions	Performance (cycles)	HSP acceleration factor versus		Conditions
		Arm® Cortex®-M33	Arm® Cortex®-M7	
Filter functions				
HSP_SEQ_Fir_f32()	6189	-	-	nbSamples = 512, numTaps = 32
	3159	x 6.6	x 4.7	nbSamples = 256, numTaps = 32
	5207	-	-	nbSamples = 256, numTaps = 64

HSP middleware functions	Performance (cycles)	HSP acceleration factor versus		Conditions
		Arm® Cortex®-M33	Arm® Cortex®-M7	
HSP_SEQ_FirDecimate_f32()	817	-	-	nbSamples = 64, numTaps = 16, decimFactor = 4
	489	-	-	nbSamples = 64, numTaps = 16, decimFactor = 8
	817	-	-	nbSamples = 128, numTaps = 8, decimFactor = 8
HSP_SEQ_Lms_f32()	37410	-	-	nbSamples = 256, numTaps = 32
HSP_SEQ_lirDf1_f32()	2838	-	-	nbSamples = 256, numTaps = 3
HSP_SEQ_lirLattice_f32()	5651	-	-	nbSamples = 256, numTaps = 3
HSP_SEQ_BiquadCascadeDf1_f32()	4260	x 3.0	x 2.7	nbSamples = 256, nBiq = 4
HSP_SEQ_BiquadCascadeDf2T_f32()	4187	x 3.0	x 2.8	nbSamples = 256, nBiq = 4
HSP_SEQ_Conv_f32()	10183	-	-	sizeA = 32, sizeB = 256
HSP_SEQ_Correlate_f32()	10190	-	-	sizeA = 32, sizeB = 256
Transform functions				
HSP_SEQ_Fft_f32()	356	x 7.7	x 5.3	log2Nbp = 6, ifftFlag = 0, bitrev = 0 or 1
	1410	x 13.0	x 8.8	log2Nbp = 8, ifftFlag = 0, bitrev = 0 or 1
	3212	-	-	log2Nbp = 8, ifftFlag = 1, bitrev = 0 or 1
	6299	x 13.9	-	log2Nbp = 10, ifftFlag = 0, bitrev = 0 or 1
HSP_SEQ_Rfft_f32()	1195	x 9.2	x 7.1	log2Nbp = 8, ifftFlag = 0, bitrev = 0 or 1
	1345	-	-	log2Nbp = 8, ifftFlag = 1, bitrev = 0 or 1
	4796	-	-	log2Nbp = 10, ifftFlag = 0, bitrev = 0 or 1
HSP_SEQ_Dct_f32()	1956	-	-	log2Nbp = 8
HSP_SEQ_IDct_f32()	2483	-	-	log2Nbp = 8
Operations on real vectors				
HSP_SEQ_VectMax_f32()	1038	x 1.8	x 1.8	nbSamples = 256
HSP_SEQ_VectCopy_f32()	273	x 2.1	x 1.1	nbSamples = 256

HSP middleware functions	Performance (cycles)	HSP acceleration factor versus		Conditions
		Arm® Cortex®-M33	Arm® Cortex®-M7	
HSP_SEQ_VectDotProd_f32()	153	x 5.9	x 5.4	nbSamples = 256
HSP_SEQ_VectCos_f32()	2573	x 6.0	x 6.0	nbSamples = 256
HSP_SEQ_VectMul_f32()	276	x 4.5	x 1.1	nbSamples = 256
HSP_SEQ_VectI2F(),	276	-	-	nbSamples = 256
HSP_SEQ_VectF2I()	1044	-	-	nbSamples = 1024
HSP_SEQ_VectQ312F(),	395	-	-	nbSamples = 256
HSP_SEQ_VectF2Q31()	1547	-	-	nbSamples = 1024
Operations on real matrix				
HSP_SEQ_MatrixInv_f32()	8661	-	-	nRows = 10, nCols = 10
HSP_SEQ_MatrixMult_f32()	19062	x 11.0	x 8.0	nRowsA,B = 32, nColsA,B = 32
HSP_SEQ_MatrixTrans_f32()	243	-	-	nRows = 10, nCols = 10
Operations on complex matrix				
HSP_SEQ_CmplxConj_f32()	527	x 2.5	x 1.5	nbSamples = 512
HSP_SEQ_CmplxDotProd_f32()	531	x 5.7	x 6.1	nbSamples = 512
HSP_SEQ_CmplxMul_f32()	530	x 5.9	x 4.5	nbSamples = 512
HSP_SEQ_CmplxMag_f32()	8717	-	-	nbSamples = 512

The following table provides the performance of basic scalar functions.

Table 9. Performance of basic scalar functions

HSP middleware functions	Description	Execution time (cycles)
Scalar function in float32		
HSP_SEQ_ScaAbs_f32()	Absolute value	3
HSP_SEQ_ScaAdd_f32(), HSP_SEQ_ScalAdd_f32() HSP_SEQ_ScaMul_f32(), HSP_SEQ_ScalMul_f32() HSP_SEQ_ScaSub_f32(), HSP_SEQ_ScalSub_f32()	Addition, subtraction or multiplication in float32	4
HSP_SEQ_ScaDiv_f32()	Division	20
HSP_SEQ_ScaSet_f32(), HSP_SEQ_ScalSet_f32()	Set a value	2
HSP_SEQ_ScaSat_f32()	Saturation	4
HSP_SEQ_ScaSqrt_f32()	Square root	18
Scalar function in integers		
HSP_SEQ_ScaAdd_i32(), HSP_SEQ_ScalAdd_i32() HSP_SEQ_ScaMul_i32(), HSP_SEQ_ScalMul_i32() HSP_SEQ_ScaSub_i32(), HSP_SEQ_ScalSub_i32()	Addition, subtraction or multiplication in integers	4
HSP_SEQ_ScaAnd_u32(), HSP_SEQ_ScalAnd_u32() HSP_SEQ_ScaOr_u32(), HSP_SEQ_ScalOr_u32() HSP_SEQ_ScaShift_i32(), HSP_SEQ_ScalShift_i32() HSP_SEQ_ScaXor_u32(), HSP_SEQ_ScalXor_u32()	Logical operations in integers	4

HSP middleware functions	Description	Execution time (cycles)
HSP_SEQ_ScaNot_u32()	Logical NOT	3
HSP_SEQ_Scalnc_u32()	Increment	2
Conditional functions		
HSP_SEQ_IfFloat(), HSP_SEQ_IfFloat() HSP_SEQ_IfInt32(), HSP_SEQ_IfInt32() HSP_SEQ_IfUInt32(), HSP_SEQ_IfUInt32()	If function	5

Revision history

Table 10. Document revision history

Date	Version	Changes
22-Avr-2026	1	Initial release.

Contents

1	General information	2
2	HSP introduction	3
2.1	What the HSP is and why it exists	3
2.2	Overview of HSP architecture	3
2.3	Operating modes	4
2.4	Features	5
2.4.1	Event-driven execution	5
2.4.2	STREAM interface	5
2.4.3	Signal processing engine (SPE)	5
2.4.4	Memory controller (MMC)	5
2.4.5	Event controller (EVTC)	5
2.4.6	Trigger input (TRGIN)	6
2.4.7	Analog-to-digital converter input (ADCIN)	6
2.4.8	Hardware signal processor DMA (HSPDMA)	6
2.5	Data types and formats (DSP and CNN)	6
2.6	BRAM organization	7
2.7	BRAM Allocation strategies by use case	7
2.7.1	DSP-only scenario	7
2.7.2	CNN-only scenario	7
2.7.3	Mixed DSP and CNN scenario	8
3	HSP middleware overview	9
3.1	Introduction	9
3.2	Middleware architecture	9
3.2.1	Components	9
3.2.2	Interaction with system DMA, interrupt services, and STEdgeAI-Core for CNN	9
3.3	Middleware services for memory management	10
3.4	Middleware options for performance and debug	10
4	Code examples and performance	11
4.1	DSP examples	11
4.1.1	Sequencer mode	11
4.1.2	Accelerator mode	14
4.1.3	Optimization tips	16
4.2	CNN acceleration using the HSP	16
4.2.1	General considerations	16
4.2.2	Set up an HSP-enabled project using STM32CubeMX	17

4.2.3	Add a model to the project using STEdgeAI-Core	18
4.3	Performance examples for a single model: tinyML® benchmarks	21
4.4	Combining DSP and CNN acceleration: streaming wake word	22
4.4.1	Use case introduction	22
4.4.2	Audio preprocessing: LFBE extraction	22
4.4.3	CNN performances and BRAM management	27
4.4.4	Combined DSP and CNN performances	29
5	Debug scenarios and how to use them	30
5.1	Break at the start of the processing list	30
5.2	Break before executing a specific processing function	31
5.3	Break on processing error	32
5.4	Asynchronous break	33
6	Trace and profiling features on the HSP	34
6.1	Time-stamp capture (TSC) block	34
6.2	Task comparators	36
6.3	Tsk_active signals	37
7	HSP DSP performance	38
	Revision history	42
	List of tables	45
	List of figures	46

List of tables

Table 1.	Applicable products	1
Table 2.	CMSIS-DSP versus HSP mapping	15
Table 3.	Comparison of inference duration on STM32U385RG with HSP and STM32U3C5ZI without HSP for three TinyML models	21
Table 4.	Per-inference energy consumption for three TinyML models	21
Table 5.	HSP performances for LFBF processing	26
Table 6.	Inference performance as a function of HSP_BRAM_AI_SIZE	27
Table 7.	Combined CNN and DSP preprocessing duration with and without HSP	29
Table 8.	Performance of middleware functions	38
Table 9.	Performance of basic scalar functions.	40
Table 10.	Document revision history	42

List of figures

Figure 1.	HSP block diagram	3
Figure 2.	Data sharing models for standard DSP functions.	7
Figure 3.	Data sharing models for standard CNN functions	8
Figure 4.	Acceleration ratio between HSP in sequencer mode and Arm® Cortex®-M33	13
Figure 5.	Energy gain ratio between HSP in sequencer mode and Arm® Cortex®-M33	13
Figure 6.	STM32CubeMX settings: enable HSP accelerator mode	14
Figure 7.	Acceleration ratio between HSP in accelerator mode and Arm® Cortex®-M33.	15
Figure 8.	Energy gain ratio between HSP in accelerator mode and Arm® Cortex®-M33	16
Figure 9.	Overview of interconnections between software components	17
Figure 10.	Setting up an HSP-enabled project using STM32CubeMX	18
Figure 11.	Structure of the project using STEdgeAI-Core.	19
Figure 12.	Inference duration with and without HSP at 96 MHz for three TinyML models	21
Figure 13.	Per-inference energy consumption for three TinyML models	22
Figure 14.	Audio pre-processing for LBFE extraction	22
Figure 15.	Buffers allocation in BRAM.	24
Figure 16.	DSP preprocessing steps in cycles	26
Figure 17.	Acceleration of HSP compared with Arm® Cortex®-M33.	27
Figure 18.	Inference duration as a function of HSP_BRAM_AI_SIZE in 32 bit words	28
Figure 19.	BRAM region size for AI configuration	28
Figure 20.	Combined CNN and DSP preprocessing duration with and without HSP	29
Figure 21.	TSC capture example	35
Figure 22.	Processing list monitoring	37

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2026 STMicroelectronics – All rights reserved