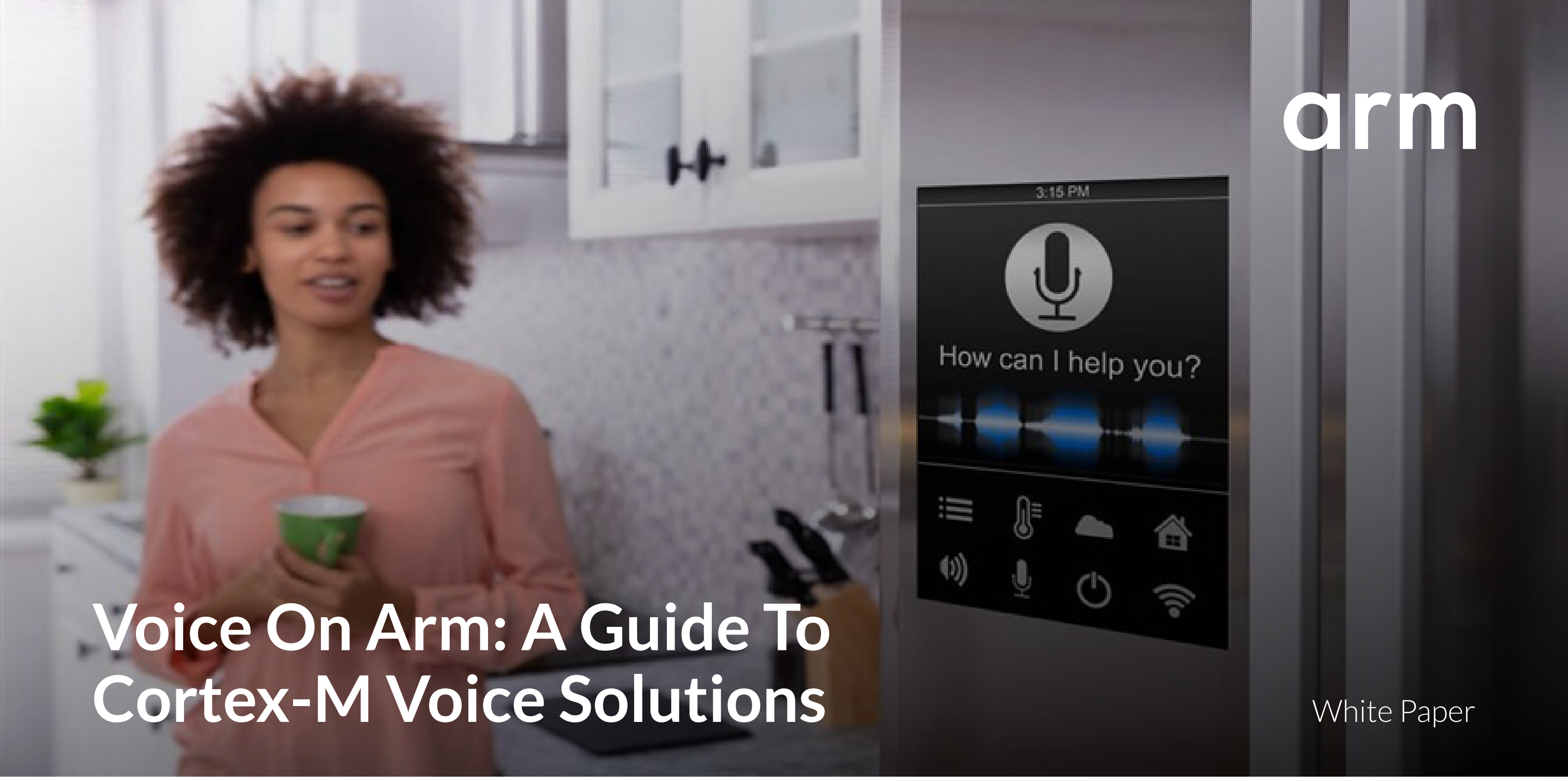


Voice On Arm: A Guide To Cortex-M Voice Solutions

White Paper

Voice User Interface

Voice interface technology and Automatic Speech Recognition (ASR) have been used in various applications for many years. The latest progress in Natural Language Understanding (NLU) and its application to voice assistants has positioned voice recognition as a key technology differentiator especially for user interfaces in consumer products. Earlier ASR solutions were limited to few word commands and may have required a specific user voice training sequence. However, modern systems that use NLU can capture various intents and phrasing and are robust to diverse accents. Deep Machine Learning (ML) applications have had a significant impact in enabling new algorithmic solutions for speech recognition and NLU.

A voice interface solution also requires efficient audio capture and rendering solutions to enable high quality user experience. Selecting the appropriate voice solution requires a good understanding of the system and computing requirements. Its implementation also requires a good understanding of the application constraints from cost to power figures.

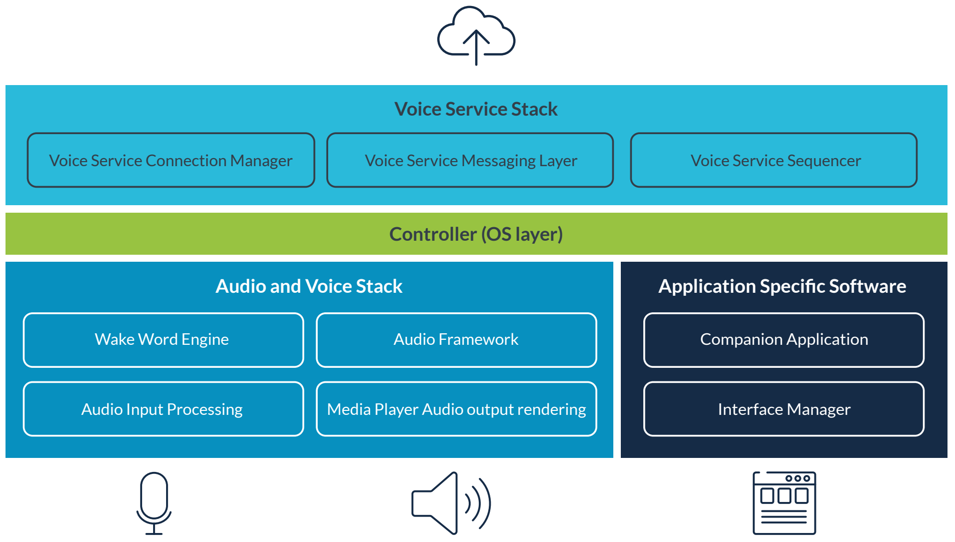
This white paper focusses on constrained voice solutions and will explore the different technology aspects that need to be understood and analyzed to achieve a suitable system architecture definition for low power, low cost voice-enabled devices.

Contents

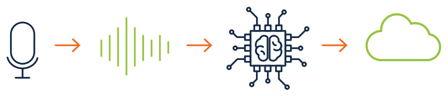
Smart Assistants

Consumers are now very familiar with voice assistants through the mass-market deployment of smart speakers. Smart speakers are one form of voice interface device, but many other types of voice-enabled consumer solutions are coming to the market. These solutions might be embedded within the product itself, or have a dedicated application form. Examples include wearable smart home appliances and automotive infotainment systems. These applications require similar technology solutions, but they may differ significantly in their usage models and integration constraints.

Most consumer voice assistant devices rely on cloud-based voice services that require a real time internet connection to an NLU server to decode user intent. These solutions are usually based on an SoC Microprocessor Unit (MPU) running a High-Level Operating System (HLOS). The HLOS hosts the voice assistant client that connects to the cloud server and the relevant software stacks for connectivity, audio, or video rendering. It also runs the dedicated device application software. The SoC MPU usually integrates the relevant system peripherals for voice capture, internet connection, and user interfaces.



While it provides broad services and great user experiences, this model is not applicable where connection to cloud-based voice services is not possible or not appropriate for cost or privacy reasons. The emergence of embedded deep learning computing solutions enables untethered NLU implementation, but sizing the dictionary and intent diversity is a complex process that has direct impact on the system computing and memory requirements.



Both models have a common need for clear user voice capture. User voice capture is primarily impacted by the acoustic and noise environment and by the speaking distance. Designing a microphone system and an associated Digital Signal Processing (DSP) system that enables speaker voice isolation from environment noise is complicated. The noise rejection level has a direct influence on the user experience as it affects the wake-word recognition rate (positive/ negative) as well as the user intent decoding (command word or NLU). The wake-word is a specific Key Word Spotting (KWS) that the user may have to say prior to any specific request. The wake-word can be used to “wake up” the speech recognition system or to interrupt any ongoing command response.

The analog voice signal must be converted to a digital format at a sampling frequency that matches the KWS and the ASR (with NLU) requirement. For communication applications, voice bandwidth is usually limited to 4KHz (8KHz sampling rate) but voice recognition may benefit from a larger bandwidth. The selected sampling frequency is a compromise between signal frequency integrity and computing resources. For example, a higher sampling rate implies higher computing needs and a larger bandwidth. Voice cloud services may also impose a specific sampling rate: 16KHz is quite common.

The acoustic signal processing and the KWS/ASR are likely to be the most demanding computing operations of the voice service embedded in the device. Breaking down the processing to their low-level DSP and ML kernels is a good way to identify the processing resource requirements.

The fundamental computing aspects to be reviewed are:

- The digital filter architecture,
- The main operation needs (dot product, convolution),
- The operand type (vector, matrix), and their coding size (8/16/32 scalar, floating point)

A first order estimation should identify raw metrics

- MAC/MHz (Multiplication and Accumulation)
- ROM/SRAM computing bandwidth (Load/Store MB/s)
- Memory density requirement

Mobile phones, smart speakers, and smart appliances usually embed significant processing capability that facilitate the implementation of the speech recognition tasks when properly powered. But wearable and IoT devices with low power budget or smart home appliances with aggressive cost requirements are usually based on more constrained systems. Maintaining a good user experience with less computing and system resources implies higher implementation complexity. We will review how an Arm Cortex-M microcontroller system with third party hardware and software can be integrated to enable a low power, low cost voice-enabled solution.

Constrained CPU System

Constrained computing systems, usually referenced as Microcontroller Units (MCUs), have fewer computing capabilities than SoC MPUs and offer a more optimized system solution. While MPUs can run HLOSs, MCUs are limited to tiny Real Time Operating Systems (RTOSs) or bare-metal software. Such configurations require much less memory (volatile and non-volatile) and less processing power while enabling efficient real time operation for control and computing tasks. These system optimizations can enable solutions with power points below 0.5 Watt and price points below $1.

Cortex-M processors

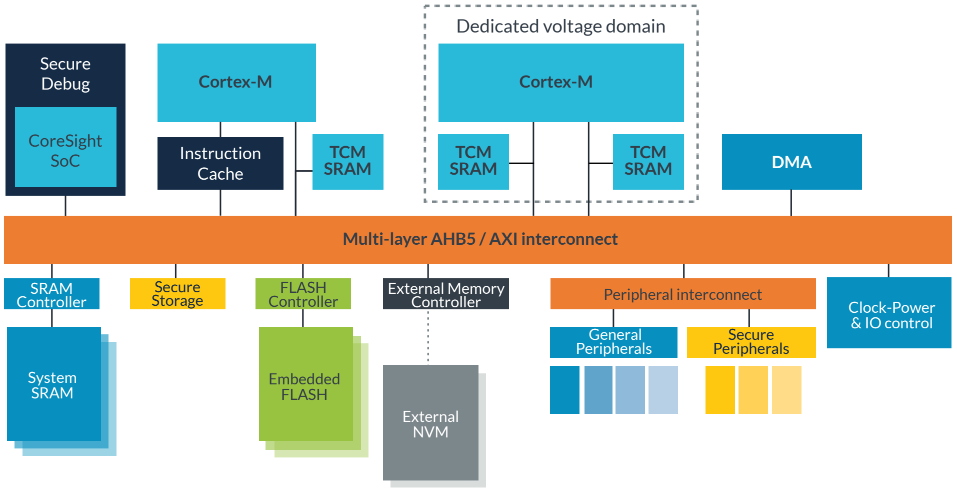
Over the years Arm has developed the Cortex-M processor family which is specifically optimized for integration in SoC MCUs. With a simpler and shorter pipeline than high-end Cortex-A processors, these 32-bit processors can still achieve raw performance (1 DMIPS/MHz) in a 100MHz -600MHz frequency range with extremely good energy efficiency. The Cortex-M family instruction set has been optimized for code density while still offering good control and compute performance with dedicated SIMD (Single Instruction Multiple Data) and DSP instructions. If you want to learn more about the Arm Cortex-M series processors please refer to [white-paper- arm info center]

Arm silicon partners offer a large portfolio of SoC MCU solutions based on the Cortex-M architecture and this paper will give some guidance for MCU platform process selection.

Microcontroller architecture

The selection of an MCU platform is mainly driven by the device application computing requirements and by the needs of a specific user interface and system integration. The voice interface itself requires particular care. This paper will go into more details regarding the voice pipeline computing requirements and the architecture implementation.

The following SoC MCU diagram depicts a generic Arm-based microcontroller architecture that is common to many Arm silicon partners. Single or multiple Cortex-M cores are attached to memory system and peripherals system via multi-layer cascaded interconnect.



Integrating multiple instances of the same Cortex-M or different Cortex-M CPUs can enable optimum computing solutions. Tuning and dedicating an instance to a specific task is an efficient way to improve the system PPA (Performance , Power, Area). Example tasks might include voice processing, security services, or user interfaces. Tuning instances in this way can significantly ease software implementation of complex tasks such as real-time low power DSP algorithms or isolation of security services. Communication between processors is facilitated by a shared memory architecture and mailbox system. Synchronization with peripherals is achieved through interrupt lines that are routed to the processors through low latency interrupt controllers based on priority and masking.

Memory system

The processor selection (Cortex-M type and frequency) is important but the SoC (embedded) memory system configuration should also be reviewed very carefully. The memory density (code and data) and the performance (latency and bandwidth) should match the requirements to execute high level and low level tasks. Memory systems with multiple hierarchy levels can be used to balance medium performance (high density, low power) and high performance (low latency, low density) using volatile and non-volatile memories. Caches or Tightly Coupled Memories (TCM) are often used to create efficient memory systems but it must be properly sized and managed. Software code profiling is suggested to analyze different task profiles (computing or latency sensitive, instruction or data intensive) and to ensure that the memory architecture matches the MCU performance requirements.

Peripheral system and interfaces

The SoC MCU should provide memory interfaces to support external memory devices when the application requires more memory than what is embedded in the SoC (most likely non-volatile storage). The external memory usage model (storage or executing in place) will dictate the memory technology (NAND, NOR) and the SoC should integrate a proper interface controller (serial, parallel, with possible error correction) to support it.

SoC MCUs offer a large variety of peripherals and interfaces which requires complex chip pinout multiplexing. After review of the application interfaces, the appropriate configuration must be verified (digital and analogue interfaces) with the compatible protocols (ports function, direction and frequency).

SoC MCU peripheral subsystems commonly integrate:

• GPIO (General Purpose Input Output)
• Timers for digital event capture, serial ports for interfaces extension (single or multi wire)
• ADC/DAC converters for analog interfaces

Pay particular attention to the interface between the external microphone and the SoC. Analogue Microphones could benefit from a SoC internal ADC converter if the ADC specifications match the microphone's audio output conversion requirement (signal dynamic and frequency). Digital Microphones could benefit from an SoC digital interface if the protocols and IO (Input-output) match the microphones interface specifications.

Microphones with a PDM (Pulse Density Modulation) interface require a sampling rate conversion (decimation filter) to get 8/16KHz audio sample rate output. It is usually preferable to perform this conversion with a dedicated hardware (filter) block with in the SoC's PDM peripheral module interface. Using the main MCU for PDM sampling rate conversion could be task intensive and less power efficient than dedicated hardware.

DMA support

Coupling a system DMA (Direct Memory Access) with peripheral subsystem could offer significant performance optimization by releasing the CPU from moving data between the interface and internal memories. Audio interfaces for microphone and speaker are good candidates for DMA management as it can enable efficient decoupling (pipelining) between the data flow management (DMA) and the data computing (CPU). System DMA offers multiple channels allowing concurrent synchronized transfers (Mem2Peripheral, Peripheral2Mem) on various interfaces (audio or communication interfaces).

Security features

Security functions are required to perform device integrity checking, device attestation, device authentication and communication encryption for the device connection needs. These security functions mainly consist of cryptography algorithms that use specific keys and protocols to protect the devices from possible attacks. The secure execution of these security functions requires dedicated hardware and software modules that form the Root of Trust (RoT) of the device. Hosting the RoT in a dedicated device (secure element device) could significantly ease the product development and the security certification (if needed) but it would increase the Bill Of Material (BOM).

MCU platforms targeting IoT applications commonly embed dedicated resources to enable RoT.

The resource list should include:

- TRNG (True Random Number Generator).
- Secure non-volatile storage for key and certificates.
- Secure boot firmware.
- Provisioning mechanism (Device Unique ID, certificate).
- Cryptographic accelerator (symmetric and asymmetric encryption).

To isolate the secure (trusted) function execution from the application (non-trusted) software a TEE (Trusted Execution Environment) should be implemented in a dedicated Cortex-M instance (if present) or on the main MCU leveraging the TrustZone (TZ) architecture available on ArmV8-M processors, such as Cortex-M33 and Cortex-M23 processors.

Power management

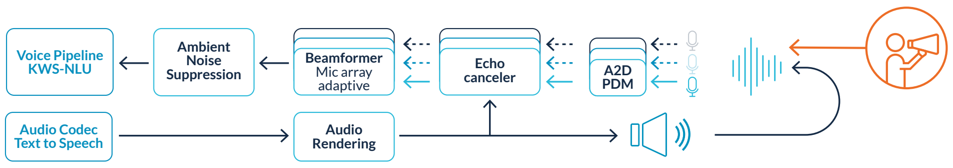
To achieve flexible integration and a low power solution the SoC MCU can implement various clock and power domains. This would include various clock generator sources (RTC, PLL) with clock routing and multiplexing function. This would also include an internal power softwareitch and internal voltage regulator to create isolated power domains dedicated to specific functions (ultra-low power always on wakeup domain, low power DSP function).

Debug management

Arm-based MCUs usually integrate efficient CoreSight debug that enable flexible Cortex-M debug capability (brake point, trace, register view) as well as SoC system visibility and control (memory image loading, clock and power control). For devices deployed in the field, the debug system should be compliant with the security policies that apply to these devices (no debug access to secure data and secure code).

Voice Capture Pipeline

The voice capture pipeline is a complex assembly of Microphone array (multiple microphones), Audio Front End (AFE) signal processing and speech recognition algorithms (KWS and ASR).


Voice Pipeline KWS-NLU
Ambient Noise Suppression
Beamformer Mic array adaptive
Echo canceler
A2D PDM
Audio Codec Text to Speech
Audio Rendering

The AFE processing is used to improve the SNR (Signal to Noise Ratio) of the user voice by applying techniques for ambient noise reduction and echo cancelation. The AFE complexity is largely dependent on the number of microphones on the microphone array and the voice capture constraints. Smart speakers usually implement multidirectional microphone arrays (up to six or seven microphones) to capture the user voice coming from any direction (360-degree mic array) at a few meters from the devices (far field capture).

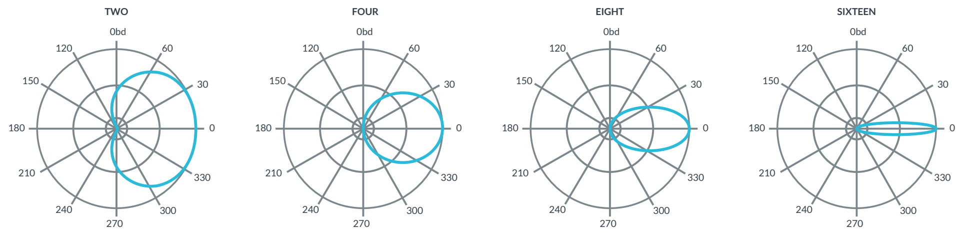
Voice capture on constrained devices targets applications where the user is likely located in front of the device within a limited solid angle at a distance that could vary from near field (around 1 meter) to far field. Depending on the user distance the microphone array can be limited to one, two or three microphones so that an MCU can still meet the AFE processing requirements.

Microphone array

Microphone technology has significantly improved since the emergence of mobile phones and MEMS (Micro-Electro-Mechanical Systems) microphones are now used in most applications thanks to their low power, integration ease and cost.

MEMS microphones are usually in the form of a tiny capsule (0.5 mm^2) with a small hole for sound entry top or bottom capsule face that can be directly soldered to a PCB (Printed Circuit Board). MEMS microphones can provide an analog or a digital signal output. Analog microphone require the addition of an appropriate analog to digital converter (ADC) while digital microphone require a digital (serial) interface for the appropriate protocol, usually Pulse Density Modulation (PDM) interface. While digital microphone are easier to integrate, thanks to the embedded ADC processing, they are usually more expensive and consume more current (~1mA) than the analog version (~0.1mA).

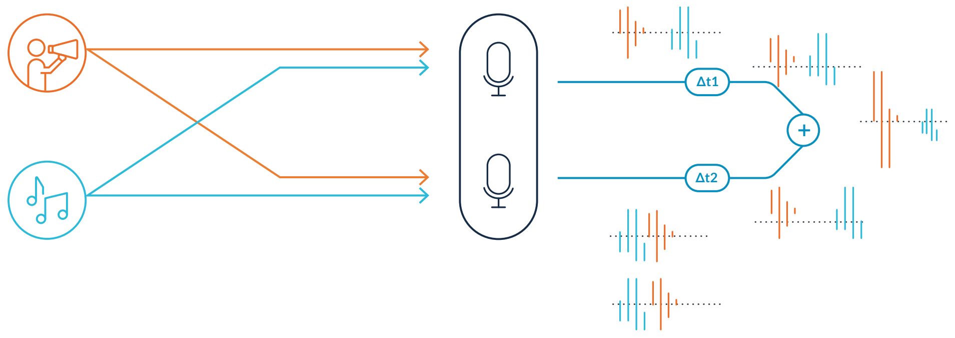
MEMS microphone are nearly omni-directional which means they capture sounds equally in every direction. Capturing the ambient sound from multiple microphone (mic array) positioned in specific ways (geometry and distance) can offer the possibility to focus the captured sound from a specific direction and to reject the sound coming from other directions. The sound propagation time induces different signal delays (phase delay) in each mic of the array that depend on the sound source localization and the mic position in the array. Combining the signals from the different microphone allow the creation of multiple focus sounds cones (beamformer) from which the sound source of interest can be extracted.



The beamforming efficiency is highly dependent on the mic's parameters disparity and more particularly the amplitude and phase matching. It is commonly recommended to use MEMS mic with a minimum of +/- 1dB matching for a SNR >60dB. The phase matching is more complex and rarely specified for mic but is important for low frequency (voice) signals where the positioning accuracy of each mic in the array is critical when the distance is reduced (<6cm). The device may incorporate a loudspeaker for the voice command feedback or other purposes. Acoustic and mechanical coupling should be reduced as much as possible and special care taken that device's loudspeaker feedback does not saturate the mic output signal.

Beamforming

Various audio beamforming architectures (fixed or adaptive direction) have been proposed in the industry with different implementation complexities. In its simple implementation a 2 mic delay and sum beamformer can be represented by the diagram below. The signal capture on a front microphone is added to the signal capture on a back microphone that is passed through a time delay stage.



By matching the acoustic steering delay stage with the time delay applied to signal capture from the mic array it can be seen that the signal from the Direction Of Arrival (DOA) is going through constructive addition compared to signal coming from the other directions (noise) that is going through destructive addition.

In this simple delay and sum form beamforming can exhibit some negative acoustic artifacts and more complex algorithms have been proposed. One of the critical aspects for far field voice detection is to address the ambient (room) reverberation that will affect both speaker voice and noise sources. The reverberation can create positive or negative signal correlation that will affect the KWS or NLU decoding. Voice signal de-reverberation is performed by digital adaptive filtering technics moving signal from time domain to frequency domain.

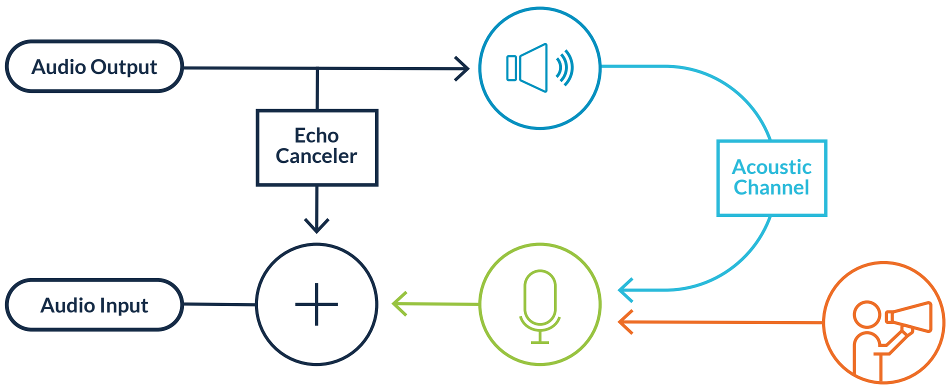
The de-reverberation may require significant processing and memory (SRAM) resources that mainly depend on the reverberation time window and the working frequency band. For voice capture in close (home) environment it is common to apply 100-200ms reverberation windows in few (4-6) frequency bands in the voice audio band (4KHz or 8KHz).

Echo canceler

Voice devices can include a speaker to provide user feedback. A device application with device audio or music would have to handle concurrent voice command capture. Play interruption (barge-in) during audio or music is a common case for smart speaker. Barge-in support usually implies the need for an echo canceler stage in the AFE to attenuate the loudspeaker's acoustic feedback captured by the microphone array.

It is likely that the loudspeaker's source will be limited to a single (mono) audio source (versus stereo common to video-audio equipment) such that the echo canceler architecture could be simplified.

Like beamformer algorithms, Acoustic Echo canceler (AEC) algorithms use delay and sum filters to suppress the reference signal (the audio feedback) from the voice signal at the microphone array. The amount of reference signal to be removed is a function of the acoustic coupling between the speaker and the mic array. This acoustic transfer function depends on device and on ambient (room) acoustic parameters, where reverberation is a critical aspect.



Similar to the adaptive beamforming, echo cancellation is based on digital adaptive filtering algorithms that convert the signal from time domain to frequency domain. The echo cancellation will probably operate in the same acoustic environment (reverberation) so that the AEC bank filter may have the same length and similar computing needs (MAC/s and SRAM) to beamforming.

Audio Front End metrics

Beamforming and AEC (when needed) can be combined in multiple ways (AEC front or back beamforming). The AFE architecture complexity and computing needs will largely depend on the mic array type and the AEC/Beamforming assembly. It is a difficult task to estimate the computing (MAC/s) and the memory (KB) requirement by just considering the number of microphone, the algorithm complexity, and the acoustic environment (voice band, reverberation).

The Arm ecosystem offers a large portfolio Audio Front Ends (AFEs) with beamforming solutions that can be tuned to 2 or 3 mic arrays. Device architects should request performance metrics from the selected AFE provider after a common review of acoustic performance needs and applicable mic array configuration.

The following metrics are average computing and memory requirement that the Arm team has measured for some of the ecosystem AFE solutions. These correspond to state-of-the-art 2 mic AFE beamforming and echo cancellation supporting 100ms echo tail for a 8KHz audio band (16KHz sampling rate). These metrics were extracted on a Cortex-M33F platform using an efficient memory system. The signal processing utilizes the Cortex-M33 FP32 instruction set.

Function	MHz	ROM	RAM
2 MIC Beamformer	60	60KB	130KB
AEC (Mono feed back echo cancellation)	70^Δ	60KB^◊	130KB^†

Average computing cycle running on Cortex-M33F and low latency memory system

^Δ AEC and Beamformer if active concurrently will not cumulate total MHz.

^◊ AEC and Beamformer shared most of the computing routine so memory needs will not cumulate.

^† AEC and Beamformer shared most of the memory (sample buffer) so memory needs will not cumulate.

Speech recognition

As mentioned earlier, voice activated devices may rely on cloud-based voice services or may rely on embedded NLU technology to run the speech recognition process and finally get the user intent.

Wake-word engine

Whether the speaker intent decoding is embedded or cloud-based the decoding process is only initiated after the recognition of a specific key word (wake-word). The wake-word spotting is a way to delineate user voice requests to the intent decoder from normal user background conversations. It prevents cloud-based devices from uploading user conversations to the cloud outside the desired command that should follow the specific wake-word.

The wake-word KWS is also used to transition the device from a low-power listening mode to a higher-power intent processing mode. For cloud based NLU, the wake-word detection initiates the connection to server and for untethered devices it could activate a larger command set KWS or it could activate a high performance NLU embedded processor. As such, most devices use the wake-word KWS as a Voice Activity Detector (VAD) running in the background.

Depending on their implementation complexity KWS can offer very strong robustness to False Accepts (FA) and False Rejects (FR), but KWS is unlikely to be considered as a very low power VAD solution. Coupling a low power front end VAD to initiate the KWS processing can further reduce power consumption. Using a VAD requires careful attention that the VAD latency does not affect the KWS behavior.

VAD algorithm

Voice Activity Detection is a critical feature for power savings during silent periods. Common VAD systems are implemented via cascaded energy detection composed of a fix hardwired detector and of a programmable detector.

The hardwired front stage detects energy on the raw samples of a single microphone from the microphone array. This front stage is usually integrated in a specific low power domain that can stay active while the rest of the SoC MPU is placed in sleep or idle mode. This low power subsystem includes the hardwired energy detector logic and the audio samples buffer. The second stage of the VAD is woken up and fed with the buffered audio sample when sound energy is detected on the front stage. The VAD software compares the input sample energy to the baseline noise energy. The signal analysis can be extended to the full microphone array and if the signal to noise ratio is confirmed during several tens of milliseconds (usual phoneme length) the MPU system wakes up and the microphone array input is processed by the AFE and KWS for wake-word recognition.

Function	MHz	ROM	RAM
VAD (software implementation)	1	10KB	10KB[Δ]
Wake-word Engine	50[◊]	256K	30KB

Average computing cycle running on Cortex-M33F and low latency memory system

[Δ] Voice buffer size is largely dependent on VAD validation window. Buffer is shared with the other AFE pipeline element

[◊] Smart speaker wake-word engine class (Strong accuracy. Compliant to consumer test suite)

KWS algorithm

Today, speech recognition techniques are using deep-learning techniques and convolutional neural networks. The input of the network is a spectral analysis made one hundred times per second. The first convolution layer is trained to extract the spectrum dynamicrophone in the peaks of energy. Those peaks of energy are associated to the way the mouth is opened and the position of the lips and tongue during the phoneme's articulation. The following layers implement the classifier operation and are trained against a large database of speech utterances with all possible variation from the people pronunciations.

The larger the neural network, the higher the probability to capture small pronunciation variations between human, with age, stress, and regional accent. The table below summarizes the accuracy, memory consumption and number of operations per second, depending on the three typical model sizes. If you want to learn more about KWS implementation on Cortex-M please refer to [https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/white-papers/the-power-of-speech].

CNN model size	Acc.	Mem.	OP/s
Small	91.6%	79KB	5M
Medium	92.2%	199KB	17M
Large	92.7%	498KB	25M

The NLU engine

An NLU engine for constrained IoT devices is likely to be dedicated to a specific target domain in which naturally spoken commands can be translated to specific actions. Target domains are relative to a specific usage like a smart appliance with its specific command set. For a specific command, the NLU engine can understand and associate various utterances. Different command (intent) to utterance mapping can be chosen but common applications with ten or so commands efficiently map to around a hundred utterances.

NLU engine architectures are based on various technologies but their implementation largely depend on the infrastructure in which the engine will execute (cloud compute versus end point device). By limiting the NLU to a specific target domain an optimized engine (model) can be executed on Cortex-M4, Cortex-M7 or Cortex-M33 with less than 100MHz budget and less than a Mbyte of non-volatile memory.

The Arm ecosystem offers various frameworks that allow user to create their own NLU engine for domain specific command set in different languages. These solutions would fit to a large set of available Cortex-M based MPUs with a minimum of 300KB for model storage (assuming 100s of domain specific utterances).

Voice-Enabled IoT System Base

Voice-activated solutions can be used in a wide range of applications and their overall system requirements may significantly differ from application to application (user interface, computing and power performances).

On top of a voice capture pipeline, a device is likely to integrate a generic communication stack (WiFi, BLE), security stack (Transport Layer Security) and audio stack (Audio codec). A Real Time Operating System (RTOS) or a simplistic bare-metal scheduler can be used to dispatch the stack's services to the upper application layer and to provide system resource access (storage and peripheral) through system drivers (software modules for hardware abstraction layer). The application layer integrates a client voice service application (for cloud-based voice service or embedded NLU framework) that will link the voice capture pipeline to the voice services through an audio framework (upstream/downstream audio buffer management and audio rendering).

RTOS

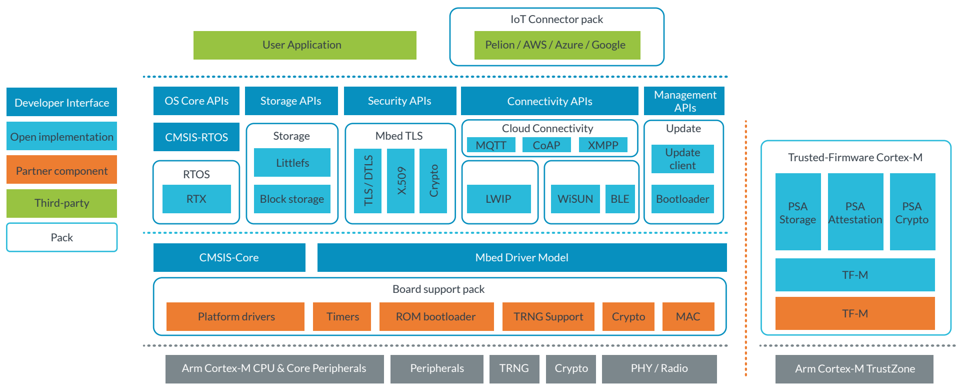
Using an RTOS is the easiest way to integrate the various software modules (voice and stack) required by the application. The software modules are broken down into multiple parallel tasks (threads) that the RTOS schedules (position for execution) based on execution priority and event requests (incoming data to be managed by the task).

The RTOS provides

- Inter-thread communication
- Interrupt dispatching
- Manages the access to resources (memory allocation, peripheral allocation).

The Arm ecosystem offers a large variety of commercial and open source RTOS for Cortex-M platforms. Many RTOSs also include pre-integrated software modules for storage, UI, power management, communication and security.

Mbed OS 6 Example Architecture



The memory footprint (Flash and SRAM) will largely depend on the added software modules required by the end application. The RTOS typically has a small footprint and requires less than 10KB of Flash and few KBs of SRAM. The total application however will require in the order of 512KB of Flash and 128KB of SRAM due to the voice pipeline and the speech recognition memory needs (see voice capture pipeline metrics).

Hardware abstraction layer (Drivers)

The Cortex-M programing model is common to all Arm-based MCUs, but the hardware system (peripherals, memory and storage) can significantly differ from SoC to SoC. It is a common development methodology to implement an interface software layer between the application and the hardware system to ease the development (debug visibility) and to offer software portability. The software application should request/access the hardware services through well define abstraction layer interface APIs (Application Programming Interface). Arm has defined a set of standard APIs for peripheral driver (CMSIS) that is widely used by the Arm MCU ecosystem.

Most of the Arm-based MCU providers deliver their MCU SoCs with a dedicated software Development Kit (SDK) that includes a Board Support Package (BSP). The BSP includes specific routines (boot routine, image loader) and all the hardware specific drivers (CMSIS compliant) to enable MCU peripheral integration (timers, DMA, I2S or PDM audio port) required for the voice application. Some SDKs also provide integrated software modules like open source RTOSs or open source Security Stacks.

Communication stack

Depending on the end application the device may have to be connected in different ways. For devices where the voice service is cloud based the communication channel requires a minimum bandwidth and real-time internet access. For devices with embedded voice service the communication bandwidth is potentially less critical and more related to device connectivity to a local network (Smart home system) or to cloud device management (Internet). The protocol for the connection to the access point (wired or wireless) should be selected considering the bandwidth requirement (Ethernet, WiFi, BLE, 802.15.4).

The protocol stack is implemented in layers (physical, network, application) that may all be executed on the voice system if supplied as an integrated solution, or distributed between the voice system and a dedicated communication device (Radio and Modem).

The application protocol is very dependent on the internet access needs, but MQTT (Message Queuing Telemetry Transport) is preferred over HTTP Transfer for constrained IoT devices. Both protocols manage the connection through the IP transport protocol. Communication limited to local connectivity may be implemented using a lighter protocol (Point to Point), but IP-based protocols are still likely to be required for local network (MQTT over IP over 802.15.4). Finally, the data exchange between the device and the network goes through the selected physical protocol stack (WiFi, BLE, 802.15.4)

Multiple communication protocol stacks for bare-metal or RTOS-integrated can be found through the Arm software ecosystem (MQTT, LwIP). The connection device (WiFi, BLE, 802.15.4) should include the relevant physical protocol stack. The following table gives some indication on the relative performance requirement and memory footprint for these software stacks.

Function	MHz	ROM	RAM
RTOS	NA	15KB$^{\Delta}$	160KB $^{\lozenge}$
Generic LIB	NA	100KB	NA$^{\lozenge}$
Communication Stack (IP-MQTT-TLS)	10 †	80KB	NA $^{\lozenge}$

Average computing cycle running on Cortex-M33F and low latency memory system

$^{\Delta}$ Average RTOS size

$^{\lozenge}$ Average RTOS heap size requirement . Most Libs , Communication stack get RAM allocated from the heap

† Average including software based AES 32KB/s encryption 8KB/s decryption.

Security stack

Device usage integrity (functionality and asset) and user privacy integrity (social and asset) are required for any IoT application. Many aspects have to be considered and it is a recommendation to run a threat analysis on the targeted application to define the device security requirements.

As previously discussed, the security foundations are based on the proper implementation of the device hardware/software RoT which all the security functions will rely on.

Device integrity is implemented through RoT services to check that the embedded software image has not been corrupted and has been securely updated (update over the air, anti-roll back mechanisms). Dynamic checking and anti-tampering techniques should be implemented to properly protect critical assets (software assets) and preserve device identity (device attestation).

The device communication channel is usually the easiest to attack so it should be properly secured in a way that preserves the integrity and privacy of the user communication. The security of the communication layers depend on the connection type (local or through Internet). Each security layer needs to be properly implemented on top of the communication channel that device communication goes through (WiFi or BL channel, Internet channel). Local connectivity protocols (WiFi, BL, 802.14.5) have their own security specification, but Internet communication supports different security protocols.

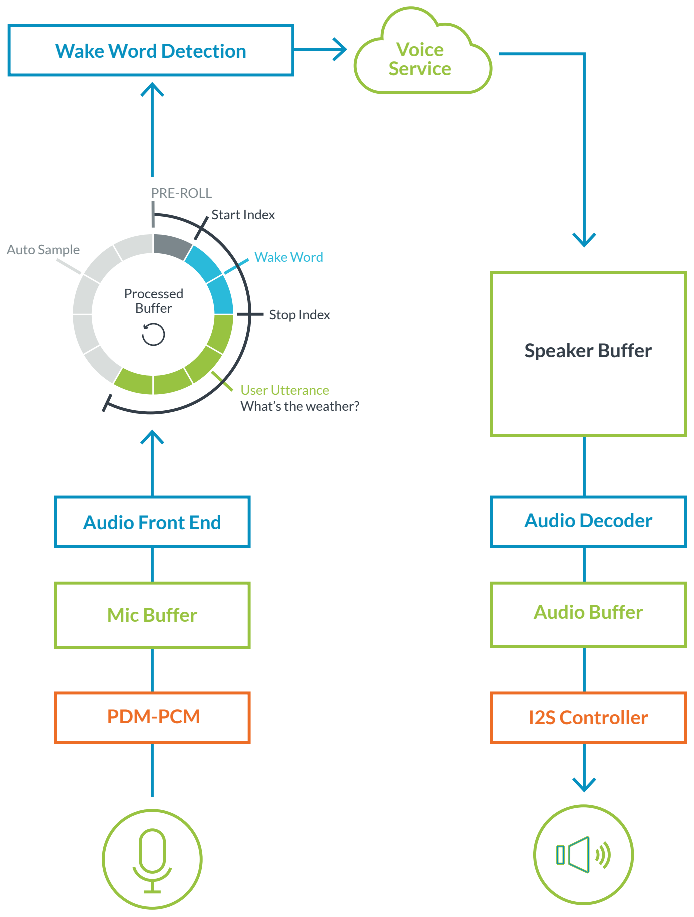
Transport Layer Security (TLS) protocol is commonly used to secure internet communication protocols like MQTT and HTTP. The TLS protocol specifies the authentication procedure between the user (device) and the server (voice service) using certificates (Certificate Authority), and also specifies the key establishment procedure to be used for communication channel encryption. The TLS security stack is integrated in many commercial and open source RTOSs with dedicated software APIs to provide access to the devices' RoT resources (certificate and physical and generated keys).

MbedTLS is an Arm Cortex-M based open source TLS stack that has been integrated in various RTOSs including MbedOS. MbedTLS includes all the cryptography algorithms and have been optimized for the Cortex-M instruction set.

Audio framework

As previously discussed a voice-controlled application requires various audio interfaces and audio computing stages to be properly work together. The audio framework includes the various buffering stages, synchronization and conversion processes required in the upstream and downstream audio paths. The audio framework complexity is largely dependent on the final application and tuned to operate with the SoC hardware resources.

Buffering is implemented between the input-output stages (mic, voice engine) and the various computing stages (AFE, KWS/NLU engine, audio decode) to adapt the processing data rate (computing windows) to the audio sampling rate. It also decouples autonomous process like DMA and compensates for CPU latency to serve asynchronous events (interrupts). Buffering introduces delays that should be properly monitored to match voice system constraints.



Conversion may be required to convert data sampling frequency (interfacing or mixing) or to convert data format. Conversion should be limited to its minimum and native matching format is preferred. The audio return path (feedback from user request) can be sent using various audio formats (raw or compressed format) and the audio framework should provide provisions (processing resources) for decoding (Audio Codec).

Function	MHz	ROM	RAM
Audio Framework	NA	NA	160KB ^Δ^
Audio decoder (Opus)	20 ◊	70KB	25KB

Average computing cycle running on Cortex-M33F and low latency memory system

Δ Sample buffer size could vary due to various implementation and content of the audio pipeline stage

◊ Audio music stream decoding Mono 16KHz – 128Kb/s

Client voice service stack

The device voice service software stack largely depends on the location of the voice service (cloud based or local device based).

Service providers for cloud-based voice services commonly provide dedicated SDKs and development support to design and integrate the client voice applications. The client stack includes procedures to connect the device (client) to the voice service account and to manage the communication between the client and the voice server. The stack also offers static and dynamic parameters to adapt to the device's specifications. The client stack relies on the audio framework to get access to the audio input-output streams. The client stack runs on top of the communication stack and on top of the security stack to get access to secure services. The client stack may also include specific application services (timer alarm) associated with voice commands.

Several cloud-based voice services released client voice SDKs for Arm based platforms and the Arm ecosystem offers many pre-integrated platform solutions. A few of these solutions have been specially designed for Cortex-M MCU platforms.

The client stack for local voice solutions largely depends on the specific voice recognition engine. For applications limited to small command vocabulary, the stack consists of a simple integration layer of a predefined command set recognition engine. Applications targeting more complex recognition solutions (Larger vocabulary and NLU for intent decoding) require a more complex framework. Various speech recognition frameworks are available through open source projects or commercial offerings. Several Arm partners have released speech recognition SDKs that generate Arm optimized solutions for Cortex-M MCU platforms.

Designing a custom Voice IoT SoC

Today Arm MCU providers offer various Cortex-M MCU solutions that are particularly tailored to enable cost effective constrained Voice IoT applications. Although these off-the-shelf solutions would serve most of the devices market needs, some applications could require custom SoC solutions. Designing a custom SoC solution would be considered when the voice interface integration into the targeted device need to address specific constrains that are not fully covered by off the shelf offering (For example combo integration, power, connectivity, or cost).

On top of Cortex-M CPU, Arm's offering also includes all major system IP that are needed to create an efficient voice-enabled SoCs.

The IP list includes:

- Interconnects infrastructure.
- Memory system (Cache, TCM and embedded memory controller).
- DMAs and specific acceleration function (Cryptographic).
- Power and Clock controllers.
- Debug infrastructure.

All these IPs are compliant to the overall Arm reference architecture that includes memory and system transaction through standard bus protocol communication, system control through standard power and clock control protocols, system development through standard system debug protocols.

To facilitate and accelerate SoC development, Arm also offers pre-defined and pre-validated Cortex-M subsystems than can be customized for integration in a standalone SoC or extended for integration in larger systems. The Arm Corstone family of subsystem series (300, 201, 102) have been specifically designed to address IoT applications leveraging Cortex-M series (M33, M23, M3) and PSA Certified principles. These subsystems include the various IP bundles, the subsystem top level integration, the subsystem verification suite and documentation.

A custom voice system would likely need IP functions (accelerator or peripheral) that are not available through Arm's IP offering but that are available through the Arm ecosystem. The selection of these complementary IPs should not only be driven by functionality requirement but also based on maturity and demonstrated compatibility with Arm architecture.

Designing a custom SoC is a complex task and requires significant expertise. ASIC design houses that are part of the Arm Approved Design Partner program can offer various services and business models that will facilitate and secure custom solution development. From specific design services to full SoC specification and production release. Arm Approved Design Partners can also facilitate full access to Arm technology (IP licensing support).

To find out more about voice recognition on Cortex-M visit:
https://developer.arm.com/solutions/internet-of-things/voice-recognition