# Avaya Experience Portal 8.1.2
# Call Classification White Paper

**Abstract**

**This paper provides information about the call classification feature in Avaya Experience Portal 8.1.2. It details the configuration and tuning of the call progress engine.**

**Issue 1.0**
**Oct 2022**

## Contents

# 1. Overview

**Avaya Experience Portal 8.1.2 incorporates a more flexible call classification feature, allowing outbound calling applications to better control the analysis of audio data. The configurability of the call classification feature has been greatly expanded to allow decisive control over the finest details of the operation of the call progress engine, enabling applications flexible control to tune and bias call classifications for their specific scenarios. This white paper provides a through description of the operation of the call progress engine, its configuration and the tools available for diagnosing failed calls.**

# 2. Theory of Operation

**When engaged on an outbound call, the call progress analysis engine receives audio data extracted from incoming RTP packets from the established media stream. The extracted audio data is first transcoded to an 8 kHz 32-bit linear floating point format to enable accurate analysis. Once transcoded, the linear samples are then organized into frames for analysis. Each frame is processed with the goal of assigning one of four general categories (SILENT, TONE, VOICE, or NOISE). This information is used to advance the state of the pattern recognition state machine. The state machine is programmed to accept certain sequences of frame categories as recognizable events, potentially invoking corresponding CCXML events in the application that initiated the outbound call.**

**Frames are equal in size to the number of samples required for the configured FFT transform and are optionally overlapped. Once the required audio data for a frame is collected, the call progress engine performs a simple time domain analysis to estimate the amount of energy in the audio frame. If the estimated energy is less than a configurable dynamic threshold then the frame is summarily categorized as SILENT without further analysis. If sufficient energy is present however, analysis continues by computing a few metrics in order to determine TONE or VOICE categorization. The first of these metrics is a more thorough time domain analysis which is made by computing the "periodicity" of the frame and measures the regularity of the pattern of the audio samples in the frame. The periodicity consists of a normalized value ranging between 0 and 1, where values closer to 1 indicate very regular patterns that repeat consistently (e.g. the periodicity of a perfectly sampled fixed frequency tone would be 1). Next, the audio data is passed through an**

**FFT algorithm to compute the power spectrum for the frame which is optionally filtered and then run through a peak detection algorithm to count the total number of significant peaks and find the one or two largest frequency components. Finally, a normalized metric named "tau", ranging between 0 and 1, is computed. Tau is the ratio of the amount of energy in the significant peaks to the total energy in the frame. The peak count, periodicity and value for tau are all used to determine if the frame's energy is consistent with a pure tone or speech. If the metrics don't conclusively match the ranges associated with a TONE or VOICE categorization, the frame is instead categorized as generic NOISE. If the TONE category is selected, further refinement is attempted by matching the largest frequency components against a configured list of "labeled" tones, enabling the call progress state machine to define cases involving certain identifiable single or dual frequency telephony tones.**

**The pattern recognition state machine processes a set of instructions arranged to match specific patterns found in a series of frame categorizations (e.g. match between 0 and 1200ms worth of VOICE frames, followed by at least 700ms worth of SILENT frames). The set of instructions can include basic logic and arithmetic statements to be performed using the set of predefined configurable parameters or a number of declared variables, allowing for high complexity in defined patterns (e.g. cycle counting, flagging, or tuning speech engine parameters based on detected patterns). Finally, the instructions for a particular pattern can include statements that will cause the Avaya Experience Portal platform to emit "connection.signal" events in the CCXML session owning the associated call.**

# 3. Configuration

**The call progress engine's overall configuration is cumulatively defined by four separate layers of data. First, the set of predefined configurable parameters are initialized with hard coded values. Since these parameters are used directly in the call progress engine, they are required to have initialized values. Next, the bulk of the configuration is loaded from a text based configuration file ($AVAYA_MPP_HOME/config/call_progress.conf). This file may include settings that will override the initial hard coded values for the predefined configurable parameters. Additionally, it contains the definitions for any declared variables, labeled tone specifications and the code providing instructions for the pattern recognition state machine. Next, values for a fixed set of the predefined parameters and a few variables declared in the default call classification configuration file are applied from the settings in the Call Progress fields in the EPM web administration,**

**as downloaded in the MPP XML configuration data. Again these settings override any previous settings. Finally, values extracted from the CCXML hints associated with initiation of the outbound call are applied. The hints can override values previously set for any of the predefined configuration parameters or declared variables on a per call basis. A CCXML hint also exists to load a named call progress configuration file such that the base configuration can be changed on a per call basis (e.g. specialized tone definitions or pattern recognition code can be used for individual calls).**

**The default call progress configuration file provided by the Avaya Experience Portal platform, installed at $AVAYA_MPP_HOME/config/call_progress.conf, is tuned to differentiate live voice speakers from a variety of answering machine greetings and provides a basic set of parameters to allow biasing classification towards either live voice or recorded messages. Control over these parameters is accomplished, on a system wide basis, by changing settings exposed through the EPM web administration interface or, on a per call basis, by injecting values from a CCXML application. More advanced adaptations are achieved by either outright replacement of the default call_progress.conf file or supplying the necessary CCXML hint to point the call progress engine to an alternate configuration file.**

## 3.1. Predefined Configuration Parameters

**The following is the list of the predefined configurable parameters. Each description includes the name used to reference the parameter when setting its value through the configuration file or a CCXML hint, its data type, hard coded value, value set in the default call_progress.conf file, valid settings and a description of its application in the call progress engine. Two data types are currently in use by the call progress engine: `uint` and `float`. A `uint` type is a 32 bit unsigned integer. The `float` type represents a 32 bit single precision IEEE floating point number.**

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| `fftOrder` | `uint` | 9 | 9 | 7-11 |
| **Determines the number of audio samples used in the FFT, as a power of 2, and, subsequently, the frame size. For example, the default value of 9 denotes $2^9 = 512$ samples.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **fftSize** | **uint** | **512** | **not set** | **None** |
| **The number of audio samples for the FFT as computed from the Value of `fftOrder`.**<br><br>**Note: Setting this property results in undefined behavior. Its only intended use is as a read only property to be used within the pattern recognition state machine code.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **shiftSize** | **uint** | **512** | **256** | **>1** |
| **The number of samples by which the audio frame is shifted before computing the next FFT. The value of 256 in the default call_progress.conf file overlaps audio frames by 50% with the associated `fftOrder` setting of 9.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **Filter** | **uint** | **0** | **0** | **0 – NONE**<br>**1 – BOX**<br>**2 – TRIANGLE**<br>**3 – PSUEDO GAUSSIAN** |
| **Specifies the type of smoothing filter kernel applied to the frequency spectrum computed from the FFT results. Filtering the frequency spectrum removes less significant peaks, leaving only the most significant frequencies for consideration by the peak detector, but diffuses the sharpness of the peaks. This potentially hampers detection of dual frequency tones when the component frequencies are relatively close (i.e. they end up in neighboring bins). Typically the default value of 0, or no filtering, provides the best results.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **K** | **uint** | **3** | **3** | **3-16** |
| **Sets the width of the smoothing filter applied to the frequency spectrum computed from the FFT results. If the value of the `filter` property is set to 0, this setting has no effect.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **deltaT** | **uint** | **2** | **3** | **>1** |
| **The number of consecutive tone frames required to recognize a tone signal.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **deltaV** | **uint** | **3** | **4** | **>1** |
| **The number of consecutive voice frames required to recognize a voice signal.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **deltanT** | **uint** | **3** | **not set** | **>1** |
| **The number of consecutive non-tone frames required to stop recognizing a tone signal.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **deltanV** | **uint** | **3** | **not set** | **>1** |
| **The number of consecutive non-voice frames required to stop recognizing a voice signal.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **cutoffGap** | **uint** | **0** | **1** | **>=0 and <cutoffVoice** |
| **Sets the minimum number of detected frequency peaks for a frame to be considered for classification as voice energy.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **cutoffVoice** | **uint** | **30** | **10** | **>cutoffGap** |
| **Sets the maximum number of detected frequency peaks for a frame to be considered for classification as voice energy.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **perSize** | **uint** | **256** | **256** | **>6 and <=min(16*minPer, fftSize)** |
| **States the number of audio samples used in the periodicity calculation. If the number of samples is less than `fftSize`, the samples used are chosen from the center of the audio frame.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| minPer | uint | 100 | 100 | >=6 and <maxPer |
| Minimum period to search. <todo> Determines the maximum frequency that will be considered a tone signal. | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| maxPer | uint | 200 | 200 | >minPer and <perSize |
| Maximum period to search. <todo> Determines the minimum frequency that will be considered a tone signal. | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| MAX_STATE_USE | uint | 2 | 10 | >=2 |
| The maximum number of times any one state can be visited when processing the pattern recognition state machine before the call progress engine will abort processing. This value places an upper bound on the processing done by the pattern recognition state machine, preventing it from looping forever. | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| traceFlags | uint | 0 | not set | Bitmask of additional tracing categories:<br>1 – energy and peak detector output<br>2 – unchanged patterns |
| Enables extremely verbose tracing on the internal processing done by the pattern recognition state machine. The extra tracing will only be logged if the Telephony trace category is set to Finest. The extra output from the energy and peak detectors can be used to determine which frames are analyzed for signals and which are skipped. The extra output from pattern recognition process is logged for every frame signal that is considered; even if the signal category is the same (i.e. the state machine remains in the current state). | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| initThreshold | float | 10.0 | 5.0 | >noiseFloor |
| Sets the initial value of the noise threshold used to determine if an audio frame is a candidate for FFT analysis. Larger values will cause the call progress engine to ignore initial audio frames unless they contain significant amounts of energy. | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **noiseFloor** | **float** | **0.5** | **0.5** | **>=0.0** |
| **Defines an absolute minimum value for the noise threshold. The noise threshold will never be allowed to settle below this number. This allows a hard minimum to be set on the energy required to perform FFT analysis on an audio frame.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **noiseOffset** | **float** | **2.0** | **1.0** | **>=0** |
| **Defines an amount of energy, relative to the current value of the noise threshold, which must be exceeded for an audio frame to be passed to the FFT analysis (i.e. the energy for the current audio frame must exceed the current value for the noise threshold plus the `noiseOffset`).** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **riseFactor** | **float** | **0.999** | **0.999** | **>0 and <1.0** |
| **Sets the coefficient used in computing the exponential moving average for the noise threshold when the current audio frame's energy is above the noise threshold. Higher values will discount older observations faster (i.e. the average will be moved closer to the instantaneous value more quickly).** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **fallFactor** | **float** | **0.95** | **0.95** | **>0 and <1.0** |
| **Sets the coefficient used in computing the exponential moving average for the noise threshold when the current audio frame's energy is below the noise threshold. Higher values will discount older observations faster (i.e. the average will be moved closer to the instantaneous value more quickly).** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| **beta** | **float** | **0.05** | **0.4** | **>=0 and <=1.0** |
| **Determines the minimum height for a peak to be detected, as a fraction of the tallest point in the frequency spectrum produced by the FFT analysis. Higher values lower the number of peaks found by the peak detector.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| delta | float | 0.1 | 0.1 | >=0 and <=2.0 |
| **Determines the "sharpness" of peaks detected by the peak detection algorithm, as measured by the sum of the differences between a point in the frequency spectrum and its left and right neighbors, as a fraction of the middle point. Higher values ensure that the peak detector only considers very sharp peaks.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| epsilon | float | 3.0 | 3.0 | >0 and <=fftSize/2 |
| **Sets the coefficient used in computing the magnitude threshold used in computing the value of tau for a given frequency spectrum. Higher values cause the tau computation to only include frequency bins with very high magnitudes.** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| toneThreshold | float | 0.95 | 0.95 | >0 and <=1.0 |
| **Sets the minimum value for tau required for an audio frame to be marked as a pure telephony tone. Higher values ensure that the majority of the energy in the frequency spectrum is contained in the detected peaks (i.e. the peaks are the only energy present in the signal, with little noise or harmonics).** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| voiceThreshold | float | 0.6 | 0.5 | >0 and <=1.0 |
| **Sets the minimum value for tau required for an audio frame to be marked as voice energy. Higher values ensure that the majority of the energy in the frequency spectrum is contained in the detected peaks (i.e. the peaks are the only energy present in the signal, with little noise or harmonics).** | | | | |

| Name | Type | Default | .conf File | Valid Values |
|---|---|---|---|---|
| dpThreshold | float | 0.95 | 0.97 | >0 and <=1.0 |
| **Sets the minimum value for the computed periodicity required for an audio frame to be marked as a pure telephony tone. Higher values for the periodicity mean that the signal consists of a specific pattern that is repeated at a fixed interval with little or no change between the iterations.** | | | | |

## 3.2. Configuration File

**A call classification configuration file is split into three separate sections named CONFIG, TONES, and PATTERNS. Each section has a unique purpose and grammar, but is dependent on the previous section and thus must always be declared in the listed order. A section is started by a line containing the section name (case sensitive), followed by one or more whitespace characters (space or horizontal tab), followed by a left curly bracket ('{'). It is closed by a line containing only a right curly bracket ('}'). Throughout a configuration file comments can be inserted; any characters on a line occurring after a pound sign ('#') will be omitted before any parsing is applied. Figure 1 contains an example of the syntax and structure used to declare the required sections of a valid call progress configuration file.**

```
# Empty configuration file template
    CONFIG {

    # Section for setting values of predefined configurable parameters or
    # defining configuration specific variables

    }

    TONES {

    # Defines frequency ranges and labels for named tones

    }

    PATTERNS {

    # Code to be executed by the pattern recognition state machine

    }
```

Figure 1 Empty configuration file template

**The CONFIG section allows for statements to assign values to the predefined configurable parameters. It also allows for statements to declare and optionally initialize configuration specific variables. Both can be referenced in the TONES and PATTERNS sections.**

**The TONES section provides frequency ranges and associated signal names to refine the categorization attributed to an audio frame if the frame is determined to contain a single or dual frequency telephony tone. The assigned signal names can be referenced in the PATTERNS section.**

**The PATTERNS section simply contains the code to be executed by the pattern recognition state machine.**

**The following parsing rules describe basic constructs referred to by rules in the CONFIG, TONE, and PATTERNS subsections that follow:**

```
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
alphanum = ALPHA / DIGIT

HCOLON = ":"
COMMA = ","
EQUAL = "="
MINUS = "-"
PLUS = "+"
WSP = %x20 / %x09

LOPEN = "("
LCLOSED = "["
ROPEN = ")"
RCLOSED = "]"

uint-val = 1*DIGIT ; decimal value between 0 and 4294967295
float-val = [ sign ] frac_const [ exp ] ; reduced to 32-bit precision
sign = "+" / "-"
frac_const = 1*DIGIT / 1*DIGIT "." *DIGIT / "." 1*DIGIT
exp = ( "e" / "E" ) [ "+" / "-" ] 1*DIGIT
```

### 3.2.1. CONFIG Section

**Assignment statements can name either a predefined configuration parameter or a variable name as declared on a preceding line. They must be followed by an equals sign and the value to be assigned. Identifiers are matched case sensitively and associated values must match the type of the parameter or variable being assigned. "`uint`" types accept only whole decimal numbers between 0 and 4294967295. "`float`" types accept arbitrary real numbers, including an optional exponent and**

**conform to a C/C++ style floating-point literal. They are however truncated to 32-bits of precision.**

**Variable declaration statements are expressed similar to assignments, but are connoted with a type specifier listed before the name identifier. Additionally, the equals sign and following value are optional and can be omitted for variables that don't require an initial value. Identifier names must be unique and cannot match any predefined configuration parameter names or previously declared variable names.**

**The following parsing rules are used in processing each line of the CONFIG section:**

```
config-line = assignment / definition
assignment = uint-assign / float-assign
definition = uint-def / float-def
uint-param = (predef-uname / identifier) ; identifier must match a
                                                    previous uint-def
EQUAL uint-val
float-assign = (predef-fname / identifier) ; identifier must match a
                                                    previous float-def
                  EQUAL float-val
uint-def = "uint" identifier [ EQUAL uint-val ]
float-def = "float" identifier [ EQUAL float-val ]
predef-uname = "fftOrder" | "shiftSize" | "filter" | "k" |
"deltaT" | "deltaV" | "deltanT" | "deltanV" |
"cutoffGap" | "cutoffVoice" | "perSize" | "maxPer" |
"minPer" | "MAX_STATE_USE" | "traceFlags"
predef-fname = "initThreshold" | "noiseFloor" | "noiseOffset" |
                  "riseFactor" | "fallFactor" | "beta" | "delta" |
                  "epsilon" | "toneThreshold" | "voiceThreshold" |
                  "dpThreshold"
identifier = (ALPHA / "_") *(alphanum / "_")
```

### 3.2.2. TONES Section

**Each line in the TONES section associates a signal name with one or two frequency ranges. The declared signal name must be unique; it must not match a signal name declared on a previous line in the TONES section. Additionally, it must not match one of the following reserved signal names: "`UNUSED`", "`SILENT`", "`NOISE`", "`TONE`", "`VOICE`", "`ANYSIG`", "`ANYTONE`", or "`CLEAR`". A frequency range is expressed using a mathematical interval notation, where each endpoint can be**

**expressed as either an FFT bin number or a numeric frequency specified in Hertz that will be converted to an equivalent FFT bin number.**

**The following parsing rules are used in processing each line of the TONES section:**

```
tone-line = signal-name HCOLON interval [ COMMA interval ]
signal-name = ALPHA *alphanum ; must be unique and cannot match a
reserved name: "UNUSED", "SILENT",
"NOISE", "TONE", "VOICE", "ANYSIG",
"ANYTONE", or "CLEAR"
interval = ( LOPEN / LCLOSED ) ( endpoint / MINUS ) COMMA
( endpoint / PLUS ) ( ROPEN / RCLOSED )
endpoint = bin-number / frequency
bin-number = 1*DIGIT
frequency = 1*DIGIT "Hz"
```

### 3.2.3. PATTERNS Section

**The code enclosed in the PATTERNS section is constructed from statements from a pseudo-assembly language. Statements can optionally define a label, used to reference that specific point in the code, but must specify exactly one instruction and affiliated operands. Instructions are executed in sequential order, although a specific set of instructions allows jumping to the labeled points in the code when conditions controlled by those instructions are met. Accordingly, a set of instructions exists to perform arithmetic operations on arbitrary data useful in designating conditions for controlling the branching of code. Collectively, these instructions provide flexibility over execution of the "`sig`" instruction. This principal instruction may branch execution if a consecutive sequence of frames for a stated signal and qualified length isn't matched. A series of "`sig`" instructions and any associated arithmetic and logic operations are used to achieve the eventual objective of alerting an application to recognized telephony events.**

**Each instruction and its effects are detailed in the subsequent descriptions. For reference, the following terms are used throughout:**

**Destination Operand – A reference by name to either a predefined configurable parameter or a declared variable of a specific type.**
**Label – A reference by name to a specific point in the code as declared in a PATTERN section statement or the predefined labels `_start` or `_end`, which point to the first declared instruction or a location immediately following the last declared instruction. If execution of the pattern recognition state machine reaches the end,**

**the pattern recognition state machine is reset and execution will again start at the first instruction once frame signal other than SILENT or NOISE is detected.**
**Source Operand – Any one of an immediate unsigned integer value, immediate floating point value or reference by name to any predefined configurable parameter or a declared variable, regardless of type.**

| Instruction | Description |
|---|---|
| `add` ***`dst`***`,` ***`src`*** | **Add *src* to *dst*.** |
| `add dst, src1, src2` | **Add *src1* and *src2* and store the result in *dst*.** |

**Performs 32-bit unsigned integer addition. Source operands are first converted to 32-bit unsigned integer values before the addition is performed. Results are then stored in the destination operand which must reference a `uint` type.**

| Instruction | Description |
|---|---|
| `addf` ***`dst`***`,` ***`src`*** | **Add *src* to *dst*.** |
| `addf dst, src1, src2` | **Add *src1* and *src2* and store the result in *dst*.** |

**Performs 32-bit floating point addition. Source operands are first converted to 32-bit floating point values before the addition is performed. Results are then stored in the destination operand which must reference a `float` type.**

| Instruction | Description |
|---|---|
| `div` ***`dst`***`,` ***`src`*** | **Divide *dst* by *src*.** |
| `div dst, src1, src2` | **Divide *src1* by *src2* and store the result in *dst*.** |

**Performs 32-bit unsigned integer division. Source operands are first converted to 32-bit unsigned integer values before the division is performed. Results are then stored in the destination operand which must reference a `uint` type.**

| Instruction | Description |
|---|---|
| `divf` ***`dst`***`,` ***`src`*** | **Divide *dst* by *src*.** |
| `divf dst, src1, src2` | **Divide *src1* by *src2* and store the result in *dst*.** |

**Performs 32-bit floating point division. Source operands are first converted to 32-bit floating point values before the division is performed. Results are then stored in the destination operand which must reference a `float` type.**

| Instruction | Description |
|---|---|
| `end` | **Jump to predefined label `_end`.** |

**Performs a jump to the predefined label `_end`. This instruction is equivalent to '`jmp _end`'.**

| Instruction | Description |
|---|---|
| `je src1, src2, label` | **Jump to *label* if *src1* is equal to *src2*.** |
| `jne src1, src2, label` | **Jump to *label* if *src1* is not equal to *src2*.** |
| `jg src1, src2, label` | **Jump to *label* if *src1* is greater than *src2*.** |
| `jge src1, src2, label` | **Jump to *label* if *src1* is greater than or equal to *src2*.** |
| `jl src1, src2, label` | **Jump to *label* if *src1* is less than *src2*.** |
| `jle src1, src2, label` | **Jump to *label* if *src1* is less than or equal to *src2*.** |

**Performs the specified comparison between the source operands and jumps to the named label if the condition is met. If the source operands are of disparate types they are first converted to 32-bit floating point values before the comparison is performed. Otherwise, either an unsigned integer or floating point comparison is applied as appropriate.**

| Instruction | Description |
|---|---|
| `jmp label` | **Unconditional jump to *label*.** |

**Performs a jump to the named label.**

| Instruction | Description |
|---|---|
| `mul dst, src` | **Multiply *dst* by *src*.** |
| `mul dst, src1, src2` | **Multiply *src1* by *src2* and store the result in *dst*.** |

**Performs 32-bit unsigned integer multiplication. Source operands are first converted to 32-bit unsigned integer values before the multiplication is performed.**

**Results are then stored in the destination operand which must reference a `uint` type.**

| Instruction | Description |
|---|---|
| `mulf` ***`dst, src`*** | **Multiply *dst* by *src*.** |
| `mulf dst, src1, src2` | **Multiply *src1* by *src2* and store the result in *dst*.** |

**Performs 32-bit floating point multiplication. Source operands are first converted to 32-bit floating point values before the multiplication is performed. Results are then stored in the destination operand which must reference a `float` type.**

| Instruction | Description |
|---|---|
| `rep event` | **Report *event* to the call progress engine.** |

**Causes the named event to be reported to the call progress engine. Events are eventually translated to specific connection.signal events in the CCXML application. One of the following predefined events must be specified:**

**R_NONE**
**R_DIAL**
**R_RING**
**R_BUSY**
**R_REOR**
**R_SIT**
**R_RCES**
**R_VOC**
**R_VCES**
**R_TONE**
**R_LIVE**
**R_ANSM**
**R_ROp**
**R_VC**
**R_NCp**
**R_IC**
**R_ROpp**
**R_NCpp**
**R_IO**
**R_FXCG**

**R_FXCD**
**R_MCES**

| Instruction | Description |
|---|---|
| `sig name interval` | **Match consecutive frames with categorization specified by *name* within the time interval specified by *interval*. If the match fails, execution jumps to the predefined `_end` label.** |
| `sig name interval, label` | **Match consecutive frames with categorization specified by *name* within the time interval specified by *interval*. If the match fails, execution jumps to the location referenced by *label*.** |

**Checks to ensure that the current sequence of consecutive frames matches the named signal and that the duration of the sequence falls within the specified time interval. State machine execution lingers on this instruction as long as all of the following are true:**

**a. Newly processed frames extend a sequence of consecutive frames (i.e. the signal categorization is the same as the previous frames in the sequence).**
**b. The duration of the sequence is less than the interval maximum**
**c. The interval maximum isn't equal to the maximum value for an unsigned integer (i.e. 4294967295).**

**If the sequence ends (i.e. signal categorization of the newly processed frame doesn't match the previous frames in the current sequence) and the duration of the sequence is within the specified time interval, the match is considered successful and execution of the pattern recognition state machine moves to the next instruction.**

**If the duration of the sequence exceeds the maximum interval time or if the sequence ends and the duration is less than the minimum interval time, then the match fails and execution instead jumps to the location referenced by the supplied label or at `_end` if a label isn't specified.**

**The signal name can be one of the predefined general categorizations ("SILENT", "TONE", "VOICE", "ANYSIG", or "ANYTONE") or reference a specific tone, by name, as declared in the TONES section.**

**The time interval is expressed using a mathematical interval notation that defines a set of whole numbers, described by a minimum and maximum value, which are expressed in terms of milliseconds. The interval notation allows flexibility in whether the set should include or exclude either the minimum or maximum value**

**(i.e. combinations of left and right, open or closed) as distinguished by the use of parentheses or square brackets. Finally, the minimum value can be declared as negative infinity and the maximum value as positive infinity by using the minus sign ('-') and plus sign ('+') respectively.**

| Instruction | Description |
|---|---|
| **`sub dst, src`** | **Subtract *src* from *dst*.** |
| **`sub dst, src1, src2`** | **Subtract *src2* from *src1* and store the result in *dst*.** |

**Performs 32-bit unsigned integer subtraction. Source operands are first converted to 32-bit unsigned integer values before the subtraction is performed. Results are then stored in the destination operand which must reference a `uint` type.**

| Instruction | Description |
|---|---|
| **`subf dst, src`** | **Subtract *src* from *dst*.** |
| **`subf dst, src1, src2`** | **Subtract *src2* from *src1* and store the result in *dst*.** |

**Performs 32-bit floating point subtraction. Source operands are first converted to 32-bit floating point values before the subtraction is performed. Results are then stored in the destination operand which must reference a `float` type.**

**The following parsing rules are used in processing each line of the PATTERN section:**

```
pattern-line = ( label HCOLON ) / [ label HCOLON ] operation
label = ( ALPHA / "_" ) *( alphanum / "_" )
operation = sig-op / rep-op / jmp-op / cond-op / uarith-op / farith-op

sig-op = "sig" 1*WSP signal time-interval [ COMMA label ]
signal = "SILENT" / "TONE" / "VOICE" / "ANYSIG" / "ANYTONE" /
signal-name
time-interval = ( LOPEN / LCLOSED ) ( time / MINUS ) COMMA
( time / PLUS ) ( ROPEN / RCLOSED )
time = 1*DIGIT ; time expressed in milliseconds

rep-op = "rep" 1*WSP event
event = "R_NONE" / "R_DIAL" / "R_RING" / "R_BUSY" / "R_REOR" /
"R_SIT" / "R_RCES" / "R_VOC" / "R_VCES" / "R_TONE" /
"R_LIVE" / "R_ANSM" / "R_ROp" / "R_VC" / "R_NCp" /
"R_IC" / "R_ROpp" / "R_NCpp" / "R_IO" / "R_FXCG" /
```

```
"R_FXCD" / "R_MCES" / "R_LOOP"

jmp-op = "end" / ( "jmp" 1*WSP label )

cond-op = ( "je" / "jne" / "jl" / "jle" / "jg" / "jge" ) 1*WSP rval
            COMMA rval COMMA label
rval = uint-val / float-val / predef_uname / predef_fname / "_time" /
            "_start_of_energy"

uarith-op = ( "add" / "sub" / "mul" / "div" )
1*WSP ( predef-uname / identifier ) ; identifier must
                                    match a previous
                                    uint-def
COMMA rval [ COMMA rval ]

farith-op = ( "addf" / "subf" / "mulf" / "divf" )
                  1*WSP ( predef-fname / identifier ) ; identifier must
                                                      match a previos
                                                      float-def
                  COMMA rval [ COMMA rval ]
```

## 3.3. EPM Downloaded Parameters

**An interface for controlling a small set of settings from the default call progress configuration file is exposed in the EPM web administration. These parameters are downloaded to each MPP as part of the normal configuration process and provide system-wide control over the associated settings. From the Call Progress section of the VoIP Settings page, values for the following fields correspond to the named predefined configurable parameters or declared variables:**

| EPM Field Name | Call Progress Identifier |
|---|---|
| Voice | voiceThreshold |
| Tone | toneThreshold |
| Periodicity | dpThreshold |
| Ring Count | RING_THRESHOLD |
| Initial (Cut Through) | CUTINIT |
| Short (Cut Through) | CUTTHR |
| Long (Cut Through) | CUT4TH |
| Initial (Max Voice) | MAXVINIT |
| Short (Max Voice) | MAXV |
| Long (Max Voice) | MAXV4TH |

## 3.4. CCXML Hints

**Using the "hint" attribute of the CCXML defined <createcall> tag, an application can control all of the configuration data used by the call progress engine on a per call basis. By passing a hint named "call_classification.inqa.config_file" a CCXML application can supply a specific base configuration file to be substituted for the default call_progress.conf file. The value for this hint should be a string with the absolute path name for the location of the configuration on the MPP file system. If this hint is specified, the referenced file is loaded in place of the default call_progress.conf file. Following this, the normal configuration procedure continues, starting with the EPM downloaded parameters being applied to the loaded configuration. Finally, the rest of the supplied hints are processed, setting any named predefined configurable parameters or declared variables. The hints for these settings follow the naming pattern "call_classification.inqa.<identifier>", where <identifier> is the name of the predefined configurable parameter or declared variable. The values of each of these hints should be string representations of their associated `uint` or `float` data. Figure 2 shows an example of the CCXML syntax used to correctly pass these hints to the CCXML <createcall> tag.**

```
…

<script>
var hintObj = new Object();
hints.call_classification = new Object();
hints.call_classification.inqa = new Object();
hints.call_classification.inqa.config_file                                    =
"/opt/Avaya/ExperiencePortal/MPP/config/mobileCPA.conf";
hints.call_classification.inqa.voiceThreshold = "0.60";
hints.call_classification.inqa.toneThreshold = "0.96";
hints.call_classification.inqa.cutoffGap = "1";
hints.call_classification.inqa.RING_THRESHOLD = "3";
</script>
<createcall … hints="hintObj"/>

…
```

Figure 2 Example CCXML excerpt showing setting of call progress hints.

# 4. Audio Debug and Tuning

**When activated, the audio debug feature provided by the call progress engine will generate a trace file for each call. The trace file will log all of the audio data and its resulting analysis and events for later review. The trace files can be accessed from the MPP service menu through a graphical viewer that displays the essential data and the resulting decisions in a timeline format. This tool is used to facilitate the tuning of the call progress engine configuration and pattern recognition state machine.**

**The audio debug feature is enabled through an Avaya Experience Portal platform parameter that can be set in one of two ways. The parameter can be permanently set through modification of the $AVAYA_MPP_HOME/config/mppconfig.xml file on an MPP, by inserting an additional <parameter> tag in the <mppsysconfig> section as illustrated:**

**<parameter name="mpp.voip.cpa.audio_debug">true</parameter>**

**The setting will then take effect the next time the MPP is restarted or rebooted and will persist as long as the setting is unchanged in the mppconfig.xml file. Alternatively, the parameter can also be controlled from the command line, using the $AVAYA_MPP_HOME/bin/config.php script. Using this method, the setting can be changed directly and with immediate effect. It will revert to its initial setting in the mppconfig.xml file if the MPP is restarted or rebooted. The command line to use is the following:**

**config.php –p mpp.voip.cpa.audio_debug=true**
**or**
**config.php –p mpp.voip.cpa.audio_debug=false**

**Once the audio debug setting is enabled, the generated trace files will appear in the MediaMgr directory, accessible through the Log option on the MPP Service Menu. The trace file names follow the format "cpadata.<MPP Call ID>".**

**Figure 3 and Figure 4 show screen captures of the audio debug trace viewer utility. The viewer is composed of three main sections. The top section is common between both figures and shows a graphical representation of the audio wave form over the time period in which audio data was processed by the call progress engine. The audio waveform is displayed in light blue. This section also contains a color coded graph of the energy estimates made for each of the processed frames; red values represent frames that did not meet the required energy threshold and green values represent frames where FFT analysis was applied. A time scale is also plotted, measured in seconds from the time when the call progress engine was engaged.**

**The middle and bottom sections each have two independent selectable tabs that can change the displayed plots. The reported events from the pattern recognition state machine can also be seen overlaid in yellow.**

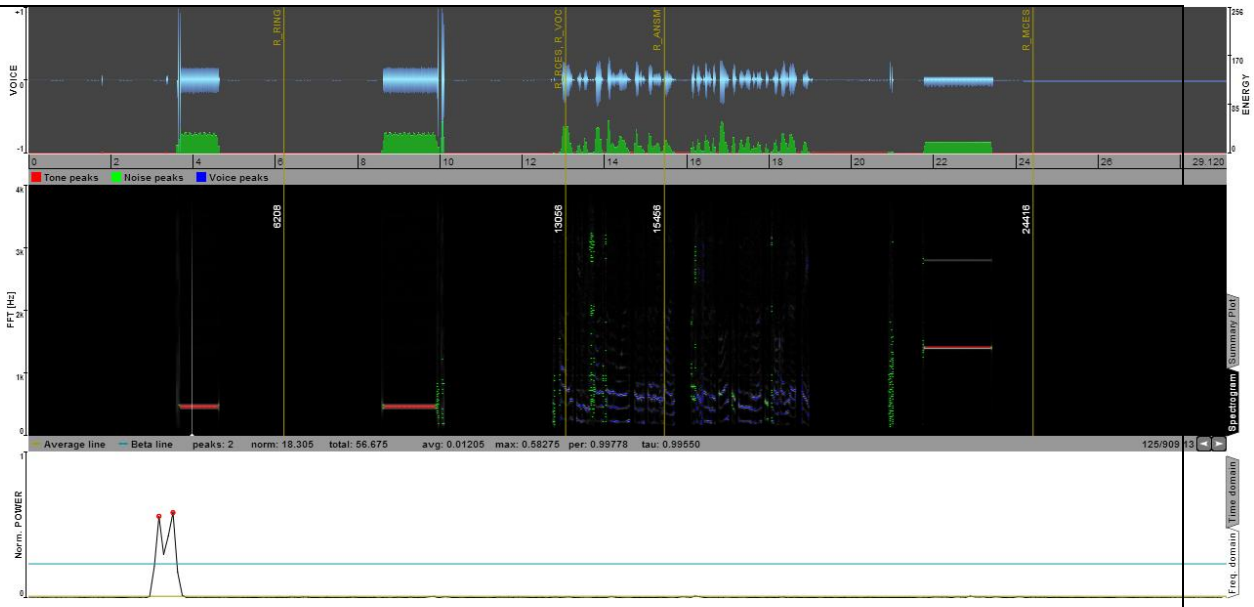


Figure 3 Audio debug tool showing Spectrogram and Frequency domain tabs.

**Figure 3 is an illustration of the data displayed when the middle section's "Spectrogram" and bottom section's "Freq. domain" tabs are selected. The data displayed by the "Spectrogram" tab is a chart of the FFT output for all analyzed audio frames plotted relative to the time scale and data in the top section. In this graph, the categorization of the audio frame and detected peaks are also visible. Red, blue and green points describe found peaks in audio frames with, respectively, TONE, VOICE and NOISE categorizations. Gray hues show FFT bin values that weren't matched by the peak detector.**

**The "Freq. domain" tab displays a horizontal plot based off the same FFT data, including the color coded found peaks, but instead for a single frame only. This tab can be used to inspect individual frames as selected by the left and right arrows in the top right corner of the bottom section or by clicking on the Spectrogram plot. The header in the top of the bottom section also displays the numeric values of the periodicity, tau and other statistics calculated for the individual frame.**

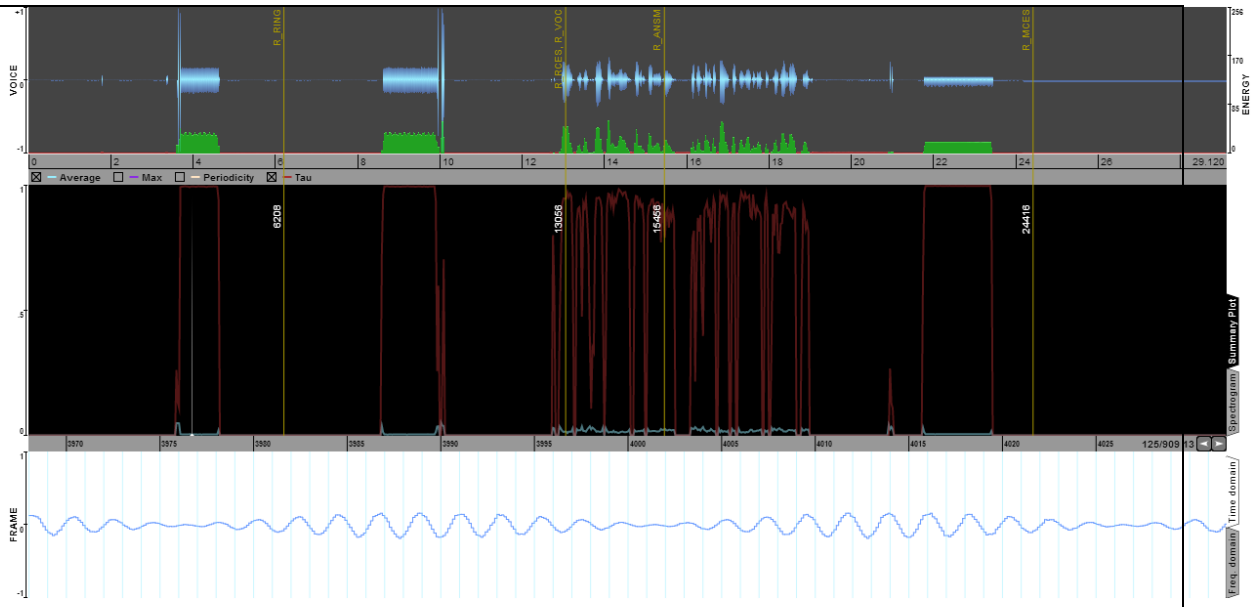


Figure 4 Audio debug tool showing Summary Plot and Time domain tabs.

**Figure 4's screen capture shows the same data with the "Summary Plot" and "Time domain" tabs selected. The "Summary Plot" tab contains a graph of the statistical values computed for each frame, plotted relative to the time scale and data in the top section. The plotted lines included in this graph can be changed by clicking the checkboxes next to the respective labels in the header of this section. The "Time domain" tab simply displays a close up view of the wave form including only the audio samples relevant for the inspected frame.**

# 5. Frequently Asked Questions (FAQ)

**How can I set “Call Progress” parameters per MPP basis having more than one MPP in EPM?**

**It is impossible. Temporary, it might be done with config.php script the following way:**

**config.php -p mpp.call_progress.inqa.max_voice.short=2750**

**It will revert if the MPP is restarted or rebooted.**

**How can I check the actual values used by single MPP in this case?**

**Using Service Menu or with xml.php script. The following “inqa” parameters match the web EPM web Call Progress page:**

**<parameter name="mpp.call_progress.inqa.threshold.voice">0.50</parameter>**
**<parameter name="mpp.call_progress.inqa.threshold.tone">0.95</parameter>**
**<parameter**
**name="mpp.call_progress.inqa.threshold.periodicity">0.97</parameter>**
**<parameter name="mpp.call_progress.inqa.threshold.rings">4</parameter>**
**<parameter name="mpp.call_progress.inqa.cut_through.init">1100</parameter>**
**<parameter name="mpp.call_progress.inqa.cut_through.short">700</parameter>**
**<parameter name="mpp.call_progress.inqa.cut_through.long">1100</parameter>**
**<parameter name="mpp.call_progress.inqa.max_voice.init">2500</parameter>**
**<parameter name="mpp.call_progress.inqa.max_voice.short">2500</parameter>**
**<parameter name="mpp.call_progress.inqa.max_voice.long">2500</parameter>**

**How does CPA count the number of ring backs?**

**The number of ring backs are counted based on the number of ring cycles detected in incoming RTP audio and it works for many countries.**

**How can I get the actual number of the ring backs detected before the classification starts?**

**The easier way is to use CPA graph.**

**How to understand what the value (Init, Short or Long) is used for Cut Through or Max Voice?**

**It depends on the number of ring back cycles detected before the classification start and the value of Ring Count. The Initial settings are applied in case a call being answered before a complete ring cycle is detected. For more details, refer to Call Progress section of *Administering Avaya Experience Portal*.**

**If the recording has two separate pauses, will the values add up or not?**

**No, Cut Through is not the sum of several gaps of the silence. Refer to *Administering Avaya Experience Portal* for more details.**

**If Max Voice is 1700, why the CPA takes longer than 1700 milliseconds to classify the recording?**

**Max Voice time is the continuous voice energy except gaps of silence shorter than the Cut Through. But there are nuances how it is calculated.**

**In general, the recording might last more than Max Voice time. The silence (as well as noise or other not definitively classified frames) becomes the part of speech voice energy if it is followed by the voice energy and the gap of silence is shorter than Cut Through.**

**For example, having Cut Through and Max Voice configured as 1100 and 1700 milliseconds appropriately. We can have 1600 milliseconds of speech energy after "start of voice" (SOV) and then 1000 milliseconds of silence after it. CPA would still not be able to classify the recording at 2600 milliseconds, the last gap of silence can not be the part of speech voice energy at this point of time. So if then the silence still continues and we reach Cut Through of 1100 milliseconds the recording will be classified as LV. But if then, instead of silence, we detect a frame of voice energy, the previous silence becomes the part of the speech voice energy and the recording will be classified as ANSM. In this case the speech voice energy will be more than 2600. It is much more than 1700 but CPA could not classify it earlier as not Cut Through nor Max Voice is reached. So theoretically the upper time level for call classification (after SOV) might be estimated as 2800 in this specific case.**

**What is the proper tool to try to figure out why a certain classification was reached?**

**CPA graph.**

**How can I find CPA trace file related to the specific session for further analysis with the graph?**

**You can look at CCXML-Session-Slot.log and find appropriate Call ID for the session.**

**@2021-03-25 13:34:31,658||FINEST|CXI|10462|Session=myxpipplylmpp01-2021084053429-46|Received CallSignal message from SessionManager:**
**type: CallSignal**
**sessionHandle: 288365492**
**connectionHandle: 305142713**
**info: [**
**MediaCtrl: "StartMedia"**
**audio.codec: "G711MULAW"**
**callID: "myxpipplylmpp01-MY_SM100-6-2021084053430"**

**In this specific case, cpadata.myxpipplylmpp01-MY_SM100-6-2021084053430 trace file relates to myxpipplylmpp01-2021084053429-46 session.**