V0.55.100

29/09/2026

HTRA API Programming Guide

HAROGIC

Contents

1. Version Management 1
2. Overview 3
3. Device and API version 5
4. Overview of Function Categories 7
5. API Call Logic and Call Map 9
   5.1 API Call Map for SWP Mode 9
   5.2 API Call Map for IQS Mode 10
   5.3 API Call Map for DET Mode 12
   5.4 API Call Map for RTA Mode 14
6. Key Variables or Settings and Concepts 16
   6.1 System 16
   6.2 Amplitude 16
   6.3 Frequency 18
   6.4 Analysis 19
   6.5 Detector and Trace Detector 20
   6.6 Default Units 22
7. Device and System Main Functions 23
   7.1 Device_List 23
   7.2 Get_APIVersion 24
   7.3 APISupportFirmwareVersions 24
   7.4 Device_ReEnumerate 25
   7.5 Device_Open 26
   7.6 Device_Close 27
   7.7 Device_QueryDeviceState/Device_QueryDeviceState_Realtime 28
   7.8 Device_QueryDeviceInfo/Device_QueryDeviceInfo_Realtime 29
   7.9 Device_QueryPowerSupplyState 30
8. Device and System Other Functions 31
   8.1 Device_SetSysPowerState 31
   8.2 Device_SetMcuSysPowerState 32
   8.3 Device_CalibrateRefClock 33
   8.4 Device_SetFanState 34
   8.5 Devcie_SetFreqResponseCompensation 35
   8.6 Devcie_SetFreqResponseCompensation_PM1 36
   8.7 Device_SetGNSSDataSource 37

8.8 Device_GetNetworkDeviceList 38
8.9 Device_SetNetworkDeviceIP 38
8.10 Device_SetNetworkDeviceIP_PM1 39
8.11 Device_GetFullUID 40
8.12 Device_GetHardwareState 40
8.13 Device_QueryDeviceInfoWithBus 41
8.14 Device_SetFreqScan 42
8.15 Device_GetFreqPlanning 43
8.16 Device_SetIFOutput 44
8.17 Device_QueryVersion_PMU 45
8.18 Device_QueryVersion_AGU 45
8.19 Device_InitIFAGC 46
8.20 Device_SetIFAGCTarget 47
8.21 Device_SetIFAGCPeriod 47
8.22 Device_ConvertEpochToReadable 48
8.23 Device_GetAmpAttenState 50
8.24 Device_QueryEIO_Version_UID 51
8.25 Device_GPIOSetBits 52
8.26 Device_GPIOResetBits 52
8.27 Device_GPIOBandSwitch 53
8.28 Device_ExtMixerConfig 54
8.29 Device_GetDecimateFactorList 55
8.30 Device_GetOnboardMemorySize 56
8.31 Device_GetBusBandwidth 57
8.32 Device_GetSupportedFunctions 58
9. System Functions Related to Device and GNSS 59
9.1 Device_SetGNSSAntennaState 59
9.2 Device_SetGNSSxpps 60
9.3 Device_SetDOCXOWorkMode 61
9.4 Device_GetGNSSInfo 61
9.5 Device_GetGNSS_SatDate/Device_GetGNSS_SatDate_Realtime 62
10. Standard Spectrum Analysis Main Functions 64
10.1 SWP_ProfileDeInit 64
10.2 SWP_Configuration 64
10.3 SWP_AutoSet 65
10.4 SWP_GetPartialSweep 66

10.5 SWP_GetFullSweep 68
11. Standard Spectrum Analysis Other Functions 70
11.1 SWP_GetPartialSweep_PM1 70
11.2 SWP_ResetTraceHold 71
12. Phase Noise Measurement Mode 72
12.1 PNM_ProfileDeInit 72
12.2 PNM_Configuration 72
12.3 PNM_StartMeasure 73
12.4 PNM_StopMeasure 73
12.5 PNM_GetPartialUpdatedFullTrace 74
12.6 PNM_AutoSearch 75
12.7 PNM_Preset_FrameDetRatio 75
12.8 PNM_Set_FrameDetRatio 76
13. Receiver/IQ Stream Main Functions 79
13.1 IQS_ProfileDeInit 79
13.2 IQS_Configuration 79
13.3 IQS_BusTriggerStart 80
13.4 IQS_BusTriggerStop 80
13.5 IQS_GetIQStream 81
14. Receiver/IQ Stream Other Functions 83
14.1 IQS_MultiDevice_WaitExternalSync 83
14.2 IQS_MultiDevice_Run 83
14.3 IQS_SyncTimer 84
14.4 IQS_GetIQStream_PM1 85
14.5 IQS_GetIQStream_PM2 86
14.6 IQS_GetIQStream_Data 87
15. Power Detection Mode 88
15.1 DET_ProfileDeInit 88
15.2 DET_Configuration 88
15.3 DET_BusTriggerStart 89
15.4 DET_BusTriggerStop 89
15.5 DET_GetPowerStream 90
15.6 DET_SyncTimer 91
16. Zero Span Mode 92
16.1 ZSP_ProfileDeInit 92
16.2 ZSP_Configuration 92

17. Real-Time Spectrum Analysis Mode 94
17.1 RTA_ProfileDeInit 94
17.2 RTA_Configuration 94
17.3 RTA_BusTriggerStart 95
17.4 RTA_BusTriggerStop 95
17.5 RTA_SetDataFormat 96
17.6 RTA_SetLookBackCmd 96
17.7 RTA_TriggerStart 97
17.8 RTA_GetIQStream 97
17.9 RTA_GetRealTimeSpectrum_Raw 99
17.10 RTA_GetRealTimeSpectrum 101
17.11 RTA_SyncTimer 102
18. MSCAN Mode 103
18.1 MSCAN_ProfileDeinit 103
18.2 MSCAN_Configuration 103
18.3 MSCAN_Start 104
18.4 MSCAN_Stop 105
18.5 MSCAN_GetData 105
19. Digital Demodulation (option) 108
19.1 Demod_Check 108
19.2 Demod_Open 108
19.3 Demod_Close 108
19.4 Demod_Reset 109
19.5 Demod_GetVersion 109
19.6 Demod_DeInit 110
19.7 Demod_Configuration 110
19.8 Demod_Execute 111
19.9 Demod_GenSymbolMap 113
20. Pulse Detection (option) 114
20.1 Pulse_Open 114
20.2 Pulse_Close 114
20.3 Pulse_Detect 115
20.4 Pulse_Detect_PM1 117
21. Auxiliary Signal Generator (option) 119
21.1 ASG_ProfileDeInit 119
21.2 ASG_Configuration 119

21.3 ASG_Configuration_WithExtMultiplier 120
22. EMI (option) 122
22.1 emi_config_scan 122
22.2 emi_get_partial_trace 123
22.3 emi_search 124
22.4 emi_config_measure 126
22.5 emi_get_measure_result 127
22.6 emi_start_measure 128
22.7 emi_stop_measure 129
22.8 emi_reset_measure 129
23. Analog Signal Demodulation Mode 131
23.1 ADM_Open 131
23.2 ADM_Close 131
23.3 ADM_AMDemod 132
23.4 ADM_FMDemod 133
23.5 ADM_AMDemod_PM1 134
23.6 ADM_FMDemod_PM1 136
24. Digital Signal Processing (Trace analysis) 139
24.1 DSP_TraceAnalysis_IM3 139
24.2 DSP_TraceAnalysis_IM2 140
24.3 DSP_TraceAnalysis_ChannelPower 141
24.4 DSP_TraceAnalysis_XdBBW 143
24.5 DSP_TraceAnalysis_OBW 144
24.6 DSP_TraceAnalysis_ACPR 146
25. Digital Signal Processing (Processing of stream) 148
25.1 DSP_Open 148
25.2 DSP_Close 148
25.3 DSP_FFT_DeInit 149
25.4 DSP_FFT_Configuration 149
25.5 DSP_FFT_IQSToSpectrum 150
25.6 DSP_DDC_DeInit 151
25.7 DSP_DDC_Configuration 152
25.8 DSP_DDC_Reset 152
25.9 DSP_DDC_GetDelay 153
25.10 DSP_DDC_Execute 153
25.11 DSP_AudioAnalysis 155

25.12 DSP_LPF_DeInit 156
25.13 DSP_LPF_Configuration 157
25.14 DSP_LPF_Reset 157
25.15 DSP_LPF_Execute_Real 158
25.16 DSP_LPF_Execute_Complex 159
25.17 DSP_InterceptSpectrum 160
26. Structure Variable 162
26.1 BootProfile_TypeDef 162
26.2 BootInfo_TypeDef 162
26.3 DeviceInfo_TypeDef 163
26.4 PowerSupplyState_TypeDef 163
26.5 DeviceState_TypeDef 163
26.6 NetworkDeviceInfo_TypeDef 163
26.7 HardWareState_TypeDef 164
26.8 GNSSInfo_TypeDef 164
26.9 GNSS_SatDate_TypeDef 165
26.10 GNSS_SNR_TypeDef 165
26.11 SWP_Profile_TypeDef 165
26.12 SWP_TraceInfo_TypeDef 168
26.13 MeasAuxInfo_TypeDef 169
26.14 SWPTrace_TypeDef 170
26.15 PNM_Profile_TypeDef 170
26.16 PNM_MeasInfo_TypeDef 171
26.17 IQS_Profile_TypeDef 171
26.18 IQS_StreamInfo_TypeDef 175
26.19 IQS/DET/RTA_TriggerInfo_TypeDef 175
26.20 IQStream_TypeDef 176
26.21 DET_Profile_TypeDef 177
26.22 DET_StreamInfo_TypeDef 178
26.23 ZSP_Profile_TypeDef 179
26.24 RTA_Profile_TypeDef 181
26.25 RTA_FrameInfo_TypeDef 183
26.26 RTA_PlotInfo_TypeDef 184
26.27 Demod_Profile_TypeDef 184
26.28 DemodInfo_TypeDef 184
26.29 Demod_SymbolMap_TypeDef 186

26.30 Pulse_Profile_TypeDef 186
26.31 PulseInfo_TypeDef 186
26.32 PulseInfoPM1_TypeDef 187
26.33 PulseTDParam_TypeDef 187
26.34 PulseAMPParam_TypeDef 187
26.35 PulseEstParam_TypeDef 188
26.36 PulseStatsParam_TypeDef 188
26.37 PulseFreqPhaseParam_TypeDef 189
26.38 MSCAN_Profile_TypeDef 189
26.39 MSCAN_Info_Typedef 189
26.40 MSCAN_Data_Typedef 190
26.41 ExtMixer_TypeDef 190
26.42 AMDemodParam_TypeDef 191
26.43 FMDemodParam_TypeDef 191
26.44 ASG_Profile_TypeDef 192
26.45 ASG_Info_TypeDef 193
26.46 emi_scan_profile 193
26.47 emi_scan_info 194
26.48 spur 194
26.49 emi_measure_profile 194
26.50 TraceAnalysisResult_IP3_TypeDef 194
26.51 TraceAnalysisResult_IP2_TypeDef 195
26.52 DSP_ChannelPowerInfo_TypeDef 195
26.53 TraceAnalysisResult_XdB_TypeDef 195
26.54 TraceAnalysisResult_OBW_TypeDef 196
26.55 DSP_ACPRFreqInfo_TypeDef 196
26.56 TraceAnalysisResult_ACPR_TypeDef 196
26.57 DSP_FFT_TypeDef 198
26.58 DSP_DDC_TypeDef 198
26.59 DSP_AudioAnalysis_TypeDef 198
26.60 Filter_TypeDef 198
26.61 DeviceFirmwareVersion_TypeDef 199
27. Structure Enumeration Variable 200
27.1 PhysicalInterface_TypeDef 200
27.2 DevicePowerSupply_TypeDef 200
27.3 IPVersion_TypeDef 200

27.4 SysPowerState_TypeDef 200
27.5 SysPowerMode_TypeDef 200
27.6 FanState_TypeDef 200
27.7 ClkCalibrationSource_TypeDef 200
27.8 GNSSDataSource_TypeDef 200
27.9 GNSSPeriphType_TypeDef 201
27.10 GNSSType_TypeDef 201
27.11 OCXOType_TypeDef 201
27.12 GNSSAntennaState_TypeDef 201
27.13 DOCXOWorkMode_TypeDef 201
27.14 SWP_FreqAssignment_TypeDef 201
27.15 Window_TypeDef 201
27.16 RBWMode_TypeDef 202
27.17 VBWMode_TypeDef 202
27.18 SweepTimeMode_TypeDef 202
27.19 Detector_TypeDef 202
27.20 TraceFormat_TypeDef 203
27.21 TraceDetectMode_TypeDef 203
27.22 TraceDetector_TypeDef 203
27.23 TracePointsStrategy_TypeDef 203
27.24 TraceAlign_TypeDef 204
27.25 FFTExecutionStrategy_TypeDef 204
27.26 RxPort_TypeDef 204
27.27 SpurRejection_TypeDef 204
27.28 ReferenceClockSource_TypeDef 204
27.29 SystemClockSource_TypeDef 205
27.30 SWP_TriggerSource_TypeDef 205
27.31 TriggerEdge_TypeDef 205
27.32 TriggerOutMode_TypeDef 205
27.33 TriggerOutPulsePolarity_TypeDef 205
27.34 GainStrategy_TypeDef 206
27.35 PreamplifierState_TypeDef 206
27.36 SWP_TraceType_TypeDef 206
27.37 LOOptimization_TypeDef 206
27.38 DSPPlatform_Typedef 206
27.39 SWPApplication_TypeDef 206

27.40 RxPort_TypeDef 207
27.41 IQS_TriggerSource_TypeDef 207
27.42 TriggerMode_TypeDef 207
27.43 TriggerTimerSync_TypeDef 207
27.44 DataFormat_TypeDef 208
27.45 DCCancelerMode_TypeDef 208
27.46 QDCMode_TypeDef 209
27.47 RBWFilterType_TypeDef 209
27.48 LookBack_TypeDef 209
27.49 IFAGC_TypeDef 209
27.50 XPPSTrigger_TypeDef 209
27.51 IQPlayBack_TypeDef 209
27.52 ASG_Port_TypeDef 210
27.53 ASG_Mode_TypeDef 210
27.54 ASG_TriggerSource_TypeDef 210
27.55 ASG_TriggerInMode_TypeDef 210
27.56 ASG_TriggerOutMode_TypeDef 210
27.57 emi_detector 210
27.58 emi_window 211
27.59 emi_unit 211
27.60 search_mode 211
27.61 Demod_FilterType_TypeDef 211
27.62 Unit_TypeDef 211
Appendix 1: API Return Index 212
Appendix 2: RTA PDF Bitmap Guide 215
Spectrum Trace Rendering 215
Render probability density as a BitMap 216

1. Version Management

Version Update Description Table

Version	Description	Date
V0.55.100	1. Added: In the Device and System Other Functions section, the following functions have been added: Device_ExtMixerConfig, Device_GetDecimateFactorList, Device_GetOnboardMemorySize, Device_GetBusBandwidth, Device_GetSupportedFunctions 2. Added: In the Auxiliary Signal Generator (option) section, the following functions have been added: ASG_Configuration_WithExtMultiplier	29/09/2026
V0.55.89	1. Added: EMI (option) section 2. Added: Firmware version information corresponding to API version 0.55.89	31/07/2026
V0.55.88	1. Added: Firmware version information corresponding to API version 0.55.88.	13/07/2026
V0.55.87	1. Added: Gaussian_CISPR has been added to Window_TypeDef	03/07/2026
V0.55.85	1. Modified: Interface changes	18/06/2026
V0.55.84	2. Added: In the Device and System section, the following functions have been added: Devcie_SetFreqResponseCompensation_PM1, Device_SetGNSSDataSource, Device_SetIFAGCPeriod	19/05/2026
V0.55.82	1. Added: In the Device and System section, the following functions have been added: APISupportFirmwareVersions, Device_ReEnumerate	13/04/2025
V0.55.79	1. Added: In the Device and System section, the following functions have been added:	11/03/2026

		Device_SetMcuSysPowerState, Device_QueryPowerSupplyState, Device_GetFreqPlanning, Device_SetIFOutput, Device_QueryVersion_PMU, Device_QueryVersion_AGU, Device_InitIFAGC, Device_GetAmpAttenState	
	2.	Refactored: Device and API version	
V0.55.77	1.	Initial Version	25/02/2026

2. Overview

This API system is implemented as a C-based dynamic link library for programming and controlling the device.

The device operating mode is the core concept of the API system. Different operating modes have different test behaviors and measurement capabilities. The first step in development is to select an appropriate operating mode according to the task requirements. The operating modes of the HTRA API system include standard spectrum analysis mode (SWP), receiver / IQ streaming (IQS), power detection (DET), real-time spectrum analysis (RTA), digital demodulation (option), MSCAN, and phase noise measurement. Fully understanding the execution mechanism of each operating mode and selecting appropriate operating modes and device parameters according to the application objectives helps fully utilize the device performance and obtain more accurate measurement results.

Device Operating Modes and Applicable Scenarios		
SWP	● Panoramic spectrum sweep ● Spectrum monitoring ● Phase noise measurement ● Harmonic measurement	● Spurious measurement ● Channel power measurement ● OBW and ACPR measurement
IQS	● Time-domain signal viewing ● IQ recording ● AM demodulation	● FM demodulation ● User application
DET	● Pulse signal observation	● Time-power relationship
RTA	● Burst signal observation ● Convert signal detection	● Dynamic spectrum observation
PNM	● Frequency stability analysis	● Close-in noise observation
Digital Demod	● Modulation quality analysis ● Vector feature analysis	● ASK/FSK/GMSK/PSK/QAM demodulation
MSCAN	● Muti-band spectrum sweep ● Segment parameter configuration	● Panorama and detail observation ● Multi-band interference check
EMI	● EMI Pre-Scan ● Disturbance Signal Search and Identification	● Limit Template Comparison ● Detailed Frequency Point Measurement

The basic API calling process consists of five steps:

1. Open the device resource;
2. According to the selected operating mode, call the corresponding category of API functions to configure the device to the specified mode;
3. Acquire measurement data using the data retrieval functions provided under the selected mode;
4. Perform user-defined analysis based on the acquired measurement data to achieve the intended application objective;
5. After the test is completed, close the device and release the associated memory resources.



Figure 1 Typical Calling Steps for SWP Mode

Before starting your application development, please carefully read the sections on API invocation logic and the API call flow chart. Using the call flow chart as the processing framework for your application will help you quickly build a robust and efficient program.

3. Device and API version

The system operates through the combined execution of multiple software and firmware components. In this system, the involved software and firmware include the device main control firmware (MFW/MCU), FPGA firmware (FFW), bus firmware (Bus), PMU firmware version (PMU), AGU firmware version (AGU), and the application programming interface (API).

Version Compatibility Principles

To ensure stable system operation, the versions of the above components must strictly follow the baseline version alignment principle. Refer to the table below for version verification and deployment.

Table 1 Software and Firmware Baseline Version Correspondence Table

API	FPGA	MCU	Bus	PMU	AGU
0.55.100	0.55.104	0.55.109	0.55.8	1.15	1.5
	0.55.30	0.55.67	0.55.8 / 0.55.2 0.55.0 / 0.47.0	-	-
0.55.89	0.55.104	0.55.104	0.55.8	1.15	1.5
	0.55.30	0.55.67	0.55.8 / 0.55.2 0.55.0 / 0.47.0	-	-
0.55.88	0.55.103	0.55.103	0.55.8	1.15	1.5
	0.55.30	0.55.66	0.55.8 / 0.55.2 0.55.0 / 0.47.0	-	-
0.55.87	0.55.103	0.55.102	0.55.8	1.15	1.5
	0.55.29	0.55.66	0.55.8	-	-
	0.55.28	0.55.65	0.55.8	-	-
0.55.84	0.55.103	0.55.99	0.55.8	1.11	1.5
	0.55.28	0.55.65	0.55.8	-	-
0.55.82	0.55.102	0.55.97	0.55.8	1.7	1.2
	0.55.28	0.55.65	0.55.8	-	-
0.55.79	0.55.100	0.55.92	0.55.8	1.7	1.2
0.55.77	0.55.99	0.55.90	0.55.6	1.7	1.2

Note:

1. To prevent system abnormalities, any form of cross-version mixing is strictly prohibited. Please ensure strict compliance with the deployment requirements.
2. The API version used must correspond to the version indicated on the cover of this manual to avoid inconsistencies between the manual description and the actual API behavior.

4. Overview of Function Categories

Table 2 Overview of API Function Categories

Function Category	Description
Device and System	Global functions that can be called under any operating mode. This category includes functions for opening and closing the device, global configuration, obtaining device information, and querying device status.
SWP	In this mode, the receiver performs frequency hopping according to the configuration to achieve frequency sweep. The baseband processes the time-domain data within the analysis bandwidth obtained at each frequency point to perform spectrum analysis and returns the spectrum results to the user. The SWP mode is suitable for measurement and analysis applications oriented toward frequency traces. This category includes functions for configuring the SWP mode, obtaining spectrum data. and trigger control.
IQS	In this mode, the receiver sets the center frequency to the specified frequency and keeps the receiver state, such as the local oscillator frequency, fixed. The baseband acquires time-domain data within the analysis bandwidth according to the specified trigger signal and returns it to the user. The IQS mode is suitable for signal recording, demodulation analysis, and multi-dimensional analysis applications. This category includes functions for configuring the IQS mode, obtaining spectrum data, and trigger control.
DET	In this mode, the receiver sets the center frequency to the specified frequency and keeps the receiver state, such as the local oscillator frequency, fixed. The baseband performs detection analysis on the time-domain signal within the analysis bandwidth and returns the power results to the user according to the specified trigger signal. The DET mode is suitable for applications that focus on the power-to-time relationship within the bandwidth, such as pulse parameter measurement. This category includes functions for configuring the DET mode, obtaining spectrum data, and trigger control.
RTA	In this mode, the receiver sets the center frequency to the specified frequency and keeps the receiver state, such as the local oscillator frequency, fixed. The baseband performs continuous spectrum analysis on the time-domain signal within the analysis bandwidth and returns the spectrum results to the user. The RTA mode is suitable for applications that focus on instantaneous and burst signals, such as interference troubleshooting and characteristic signal identification in complex electromagnetic environments. This category includes functions for configuring the RTA mode, obtaining spectrum data, and trigger control.
ASG	Global functions for controlling the device or its analog signal source options, which can be invoked under any operating mode. This category includes functions for configuring output tones, frequency sweeps, and similar operations.

DSP	General post-processing functions that are independent of hardware status. This category includes functions for IQ data processing, such as DDC, FFT analysis, and video detection; as well as measurement and analysis of spectrum traces, including IM3, phase noise, channel power, occupied bandwidth, and similar functions.
ADM	Analog demodulation post-processing functions that are independent of hardware status. This category performs demodulation on IQ data acquired by the device, including functions such as AM demodulation and FM demodulation.
PNM	In this mode, the receiver performs phase noise measurement and analysis on the acquired data, and outputs results in real time, including carrier characteristics and phase noise traces.
Digital Demod	IQS mode data post-processing functions that are independent of hardware status. This category performs demodulation on IQ data acquired by the device and returns demodulation results, constellation diagrams, EVM, and other related parameters.
Pulse Detect	IQS/DET mode data post-processing functions that are independent of hardware status. This category performs pulse detection on the acquired time-domain data and can output parameters such as pulse width, period, and duty cycle.
MSCAN	In MSCAN mode, the spectrum analyzer sequentially scans each frequency band and outputs the measurement results. It is suitable for applications such as multi-band monitoring, interference detection, and automated testing. MSCAN is a multi-segment spectrum scanning function that allows multiple frequency ranges to be scanned in a single measurement. Each scan segment can be configured independently with measurement parameters such as reference level, decimation factor, and FFT size, enabling users to simultaneously observe the spectrum and acquire IQ data, thereby improving measurement efficiency.
EMI	EMI pre-scan and spot-frequency measurement are implemented based on the Sweep and IQS operating modes. EMI supports multiple detector types, limit-line comparison, and over-limit signal searching. It enables fast wideband pre-scanning, automatic localization of abnormal signals, and accurate Peak, RMS, Linear Average, and Quasi-Peak measurements, making it suitable for EMI pre-compliance testing, troubleshooting, and EMC compliance applications.

5. API Call Logic and Call Map

5.1 API Call Map for SWP Mode

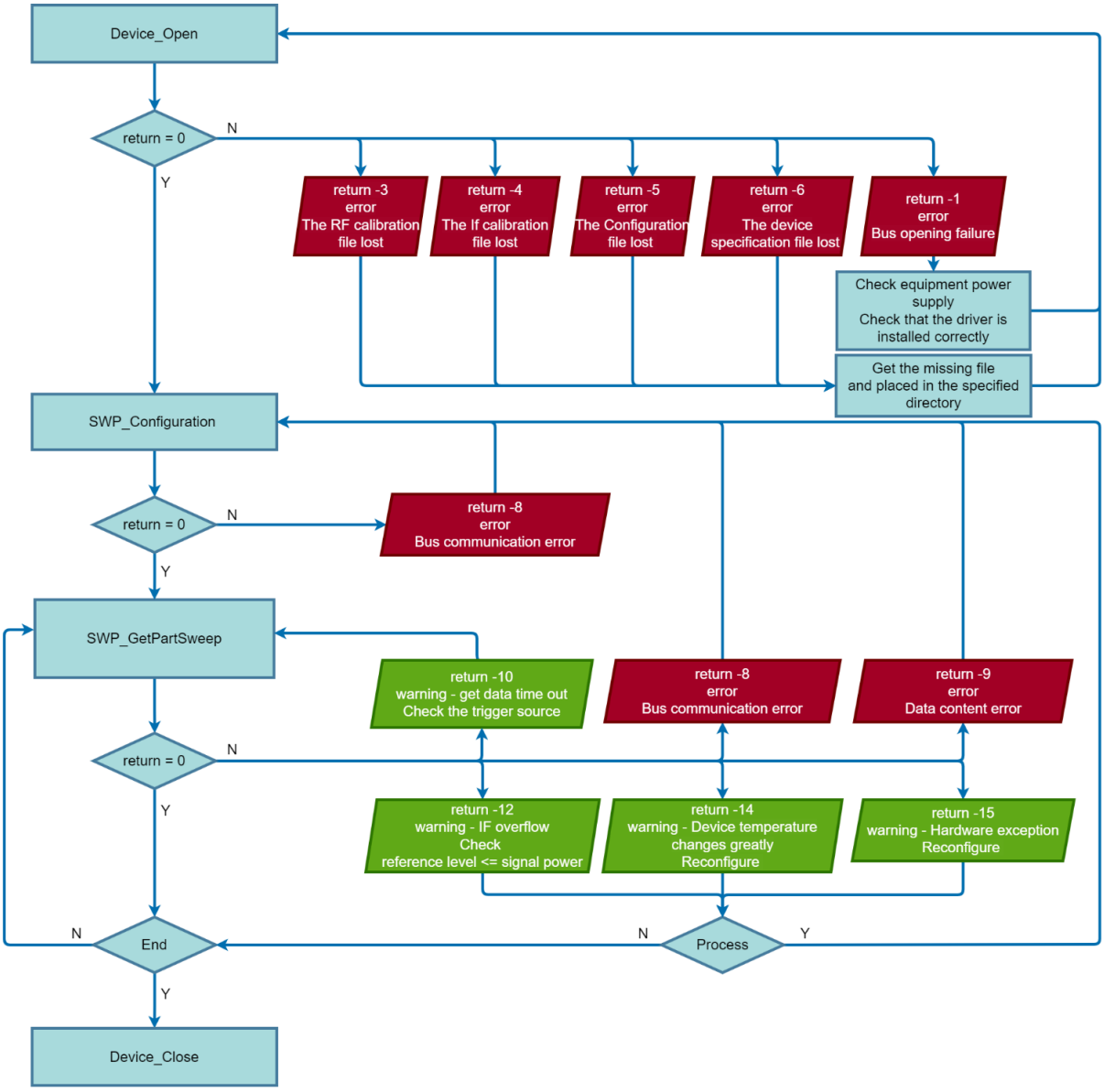


Figure 2 SWP Mode Call Flow Diagram

5.2 API Call Map for IQS Mode

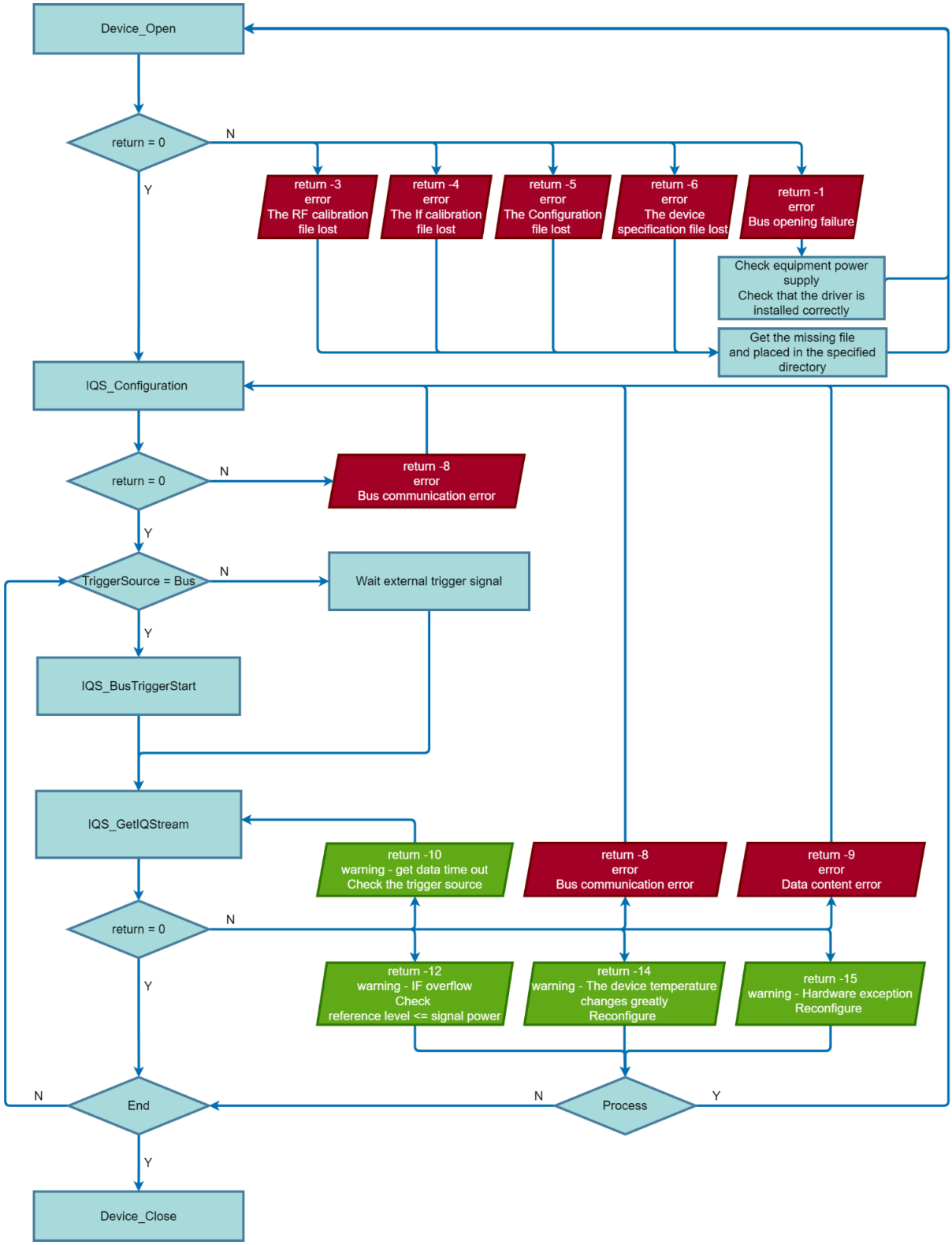


Figure 3 IQS Mode Call Flow Diagram (Trigger Mode is Fixed)

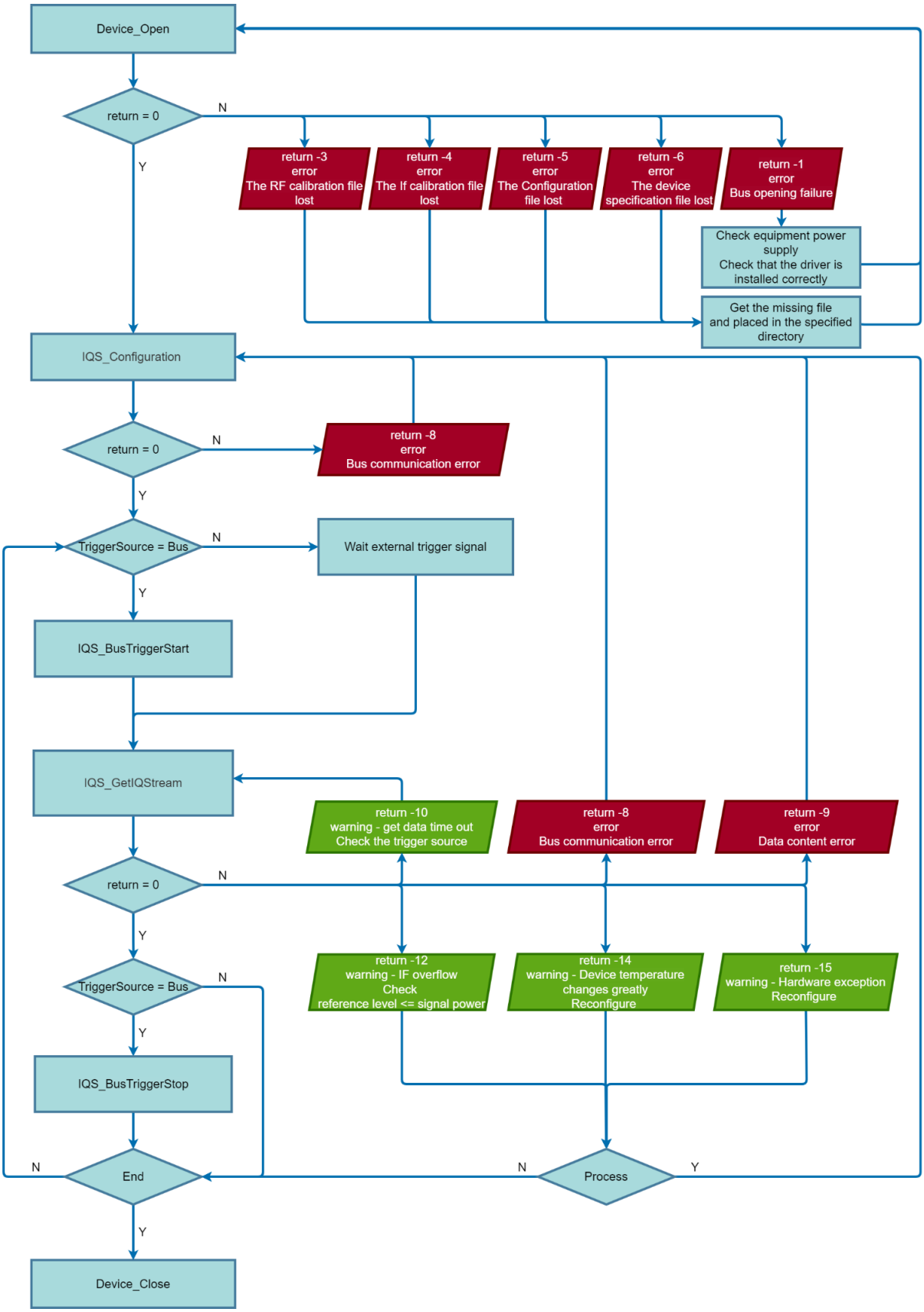


Figure 4 IQS Mode Call Flow Diagram (Trigger Mode is Adaptive)

5.3 API Call Map for DET Mode

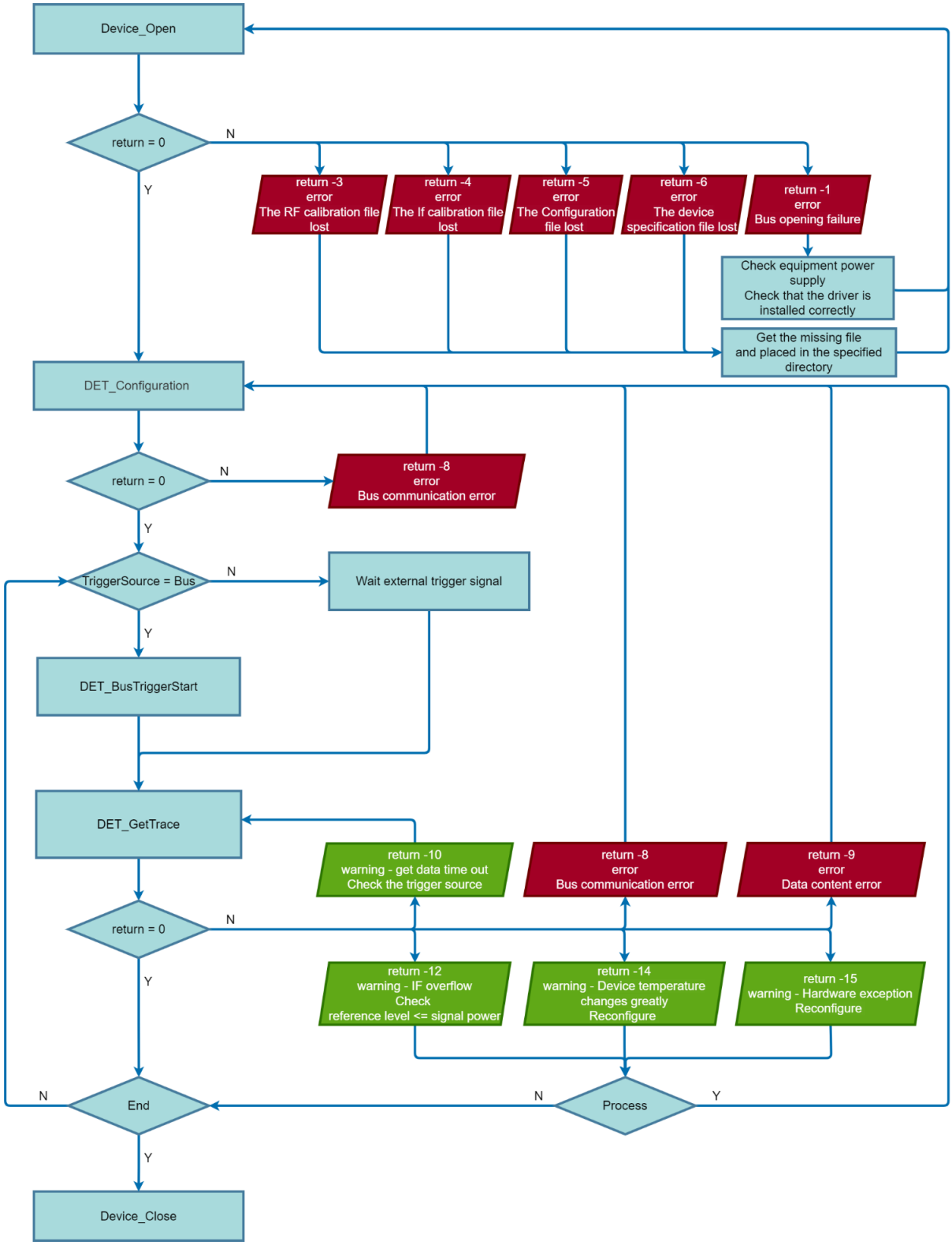


Figure 5 DET Mode Call Flow Diagram (Trigger Mode is Fixed)

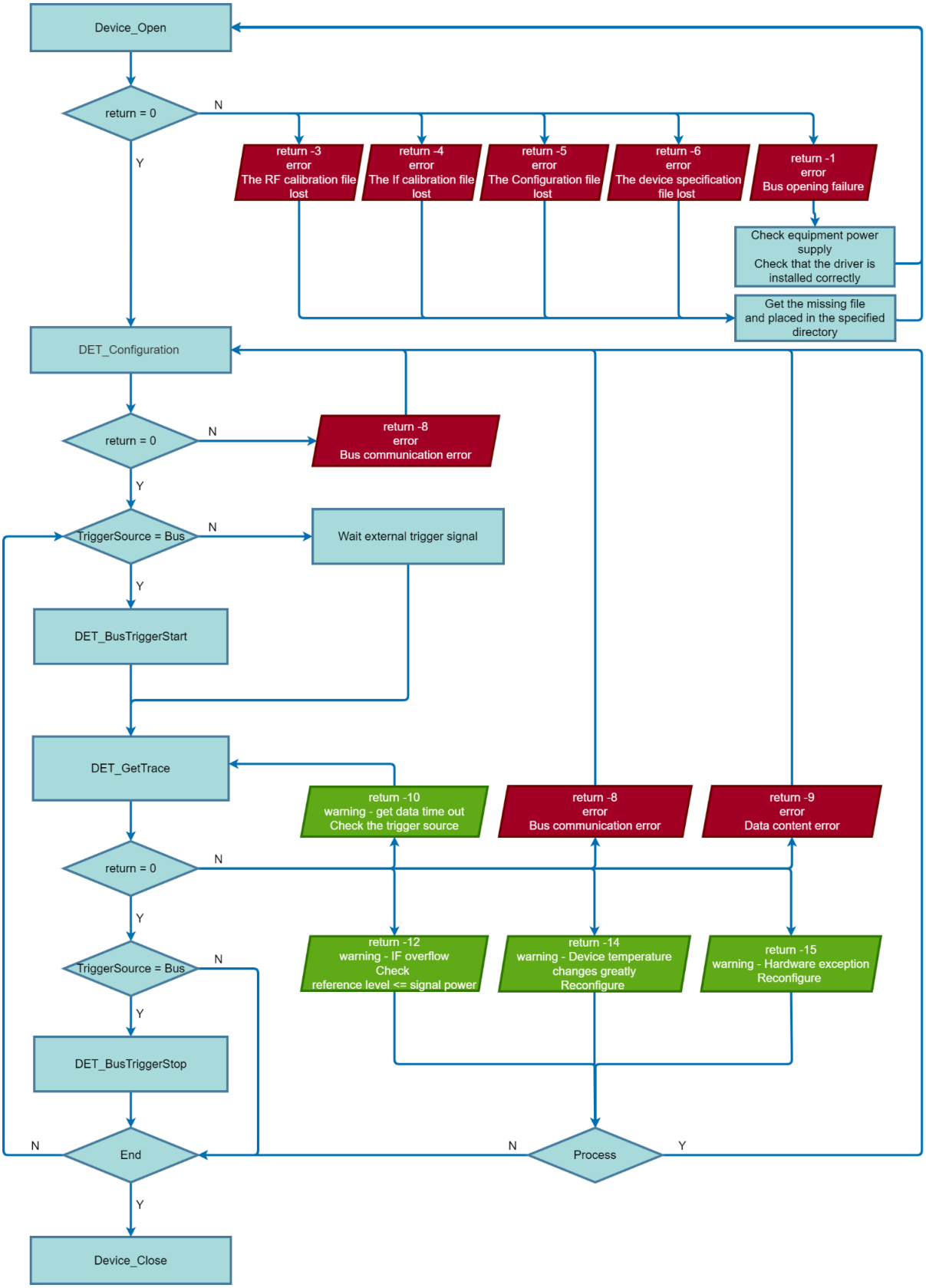


Figure 6 DET Mode Call Flow Diagram (Trigger Mode is Adaptive)

5.4 API Call Map for RTA Mode

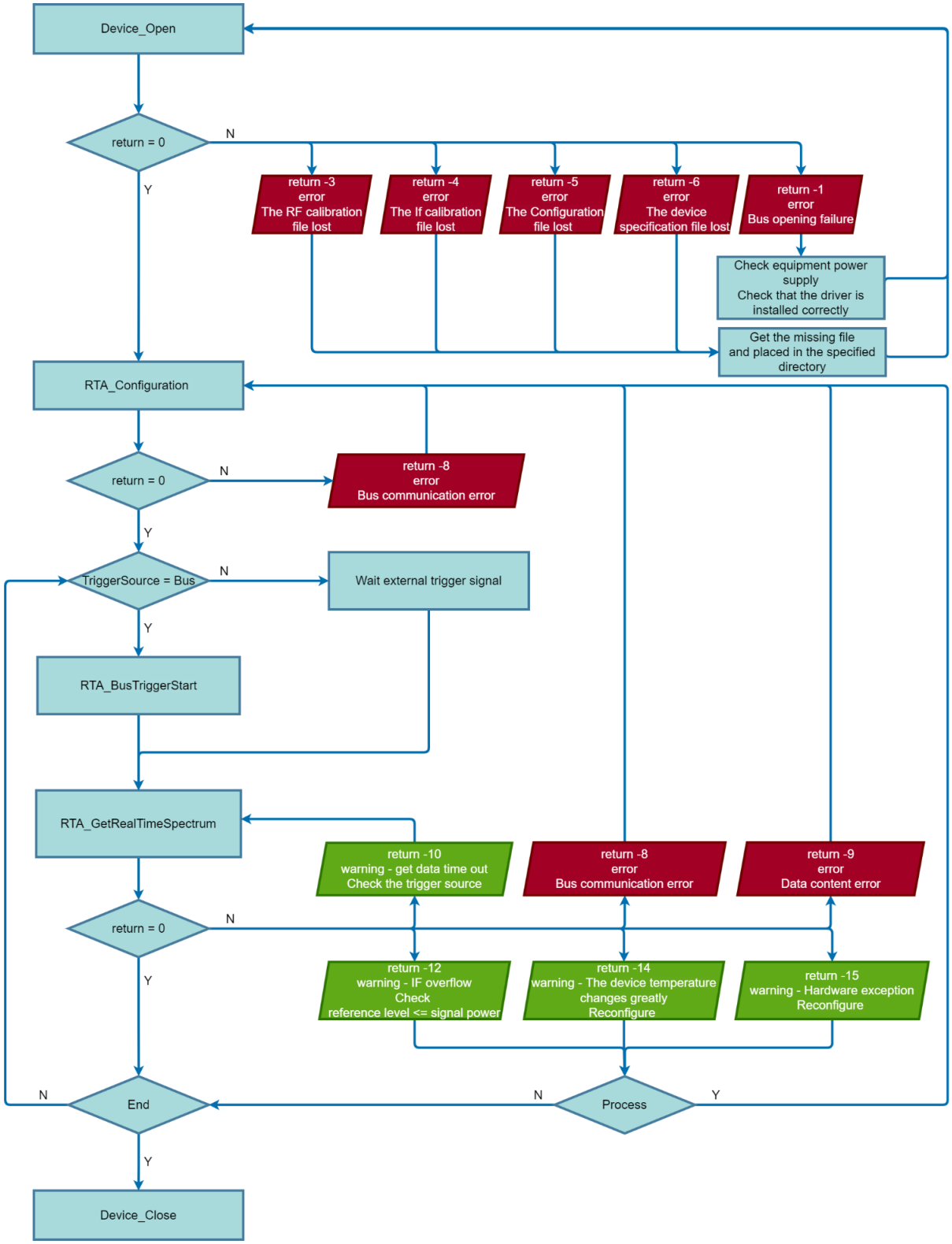


Figure 7 RTA Mode Call Flow Diagram (Trigger Mode is Fixed)

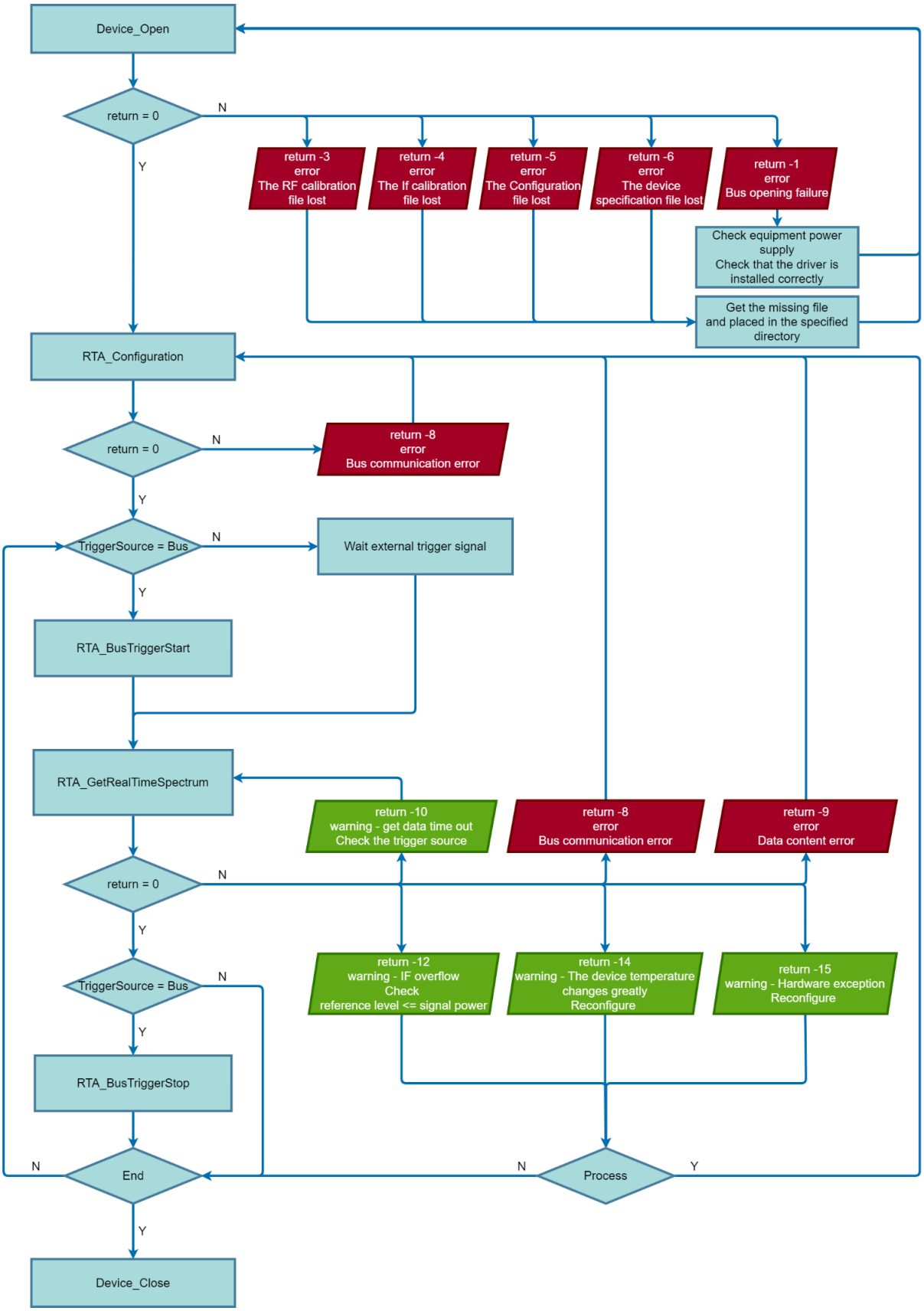


Figure 8 RTA Mode Call Flow Diagram (Trigger Mode is Adaptive)

6. Key Variables or Settings and Concepts

This chapter lists some of the key parameters and concepts related to the spectrum analyzer/receiver products. A thorough understanding of these parameters and concepts is essential for the correct and efficient use of the equipment. This summary is provided for quick reference and convenient consultation when issues arise.

6.1 System

Table 3 Table of System Parameters and Related Concept Descriptions

No.	Parameters	Applicable	Description
1	Void** Device	Device/SWP/ IQS/DET/RTA	This parameter serves as a memory reference required for device operation. When calling the API, this reference must be used to index the currently opened device.
2	DeviceUID	Device	Each device has a unique device ID, which should be used to distinguish between different device units

6.2 Amplitude

Table 4 Table of Amplitude Parameters and Related Concept Descriptions

No.	Parameters	Applicable	Description
1	RefLevel_dBm	SWP/IQS/ DET/RTA	By default, the system automatically configures the attenuator and preamplifier based on the reference level. The reference level can be simply understood as the maximum input power that the system can accept without saturation. When handling the reference level, the system maintains a certain margin, typically 1 to 6dB. Therefore, at some frequencies, even if the input power exceeds the reference level, the system may not report a saturation warning; this is normal. To achieve optimal dynamic range, set the reference level slightly above the expected maximum input signal power. For example, if the expected signal is a -3dBm tone, setting the reference level to 0dBm allows for good observation of the signal dynamics
2	Atten	SWP/IQS/ DET/RTA	The attenuation defaults to automatic mode (Atten = -1), in which case the system sets the channel attenuation solely based on the reference level. To manually configure the channel attenuation, set Atten to the desired value.
3	Preamplifier	SWP/IQS/ DET/RTA	For devices equipped with a preamplifier, enabling or disabling the preamplifier significantly affects the system's noise performance and linearity: 1). Preamplifier enabled: Reduces system noise but lowers the maximum linear input power and the

			maximum damage threshold; 2). Preamplifier disabled: Increases the allowable input power range, but system noise will be relatively higher. The system can typically be configured to: 1). Automatically control the preamplifier based on the reference level; 2). Force the preamplifier off to prevent potential damage from overload.
4	AnalogIFBW Grade	SWP/IQS/ DET/RTA	For devices equipped with multiple analog intermediate-frequency (IF) filters, the system provides several IF channels with different characteristics for selection. Different analog IF bandwidths offer varying out-of-band suppression, in-band flatness, group delay, and other performance metrics. Please choose the appropriate IF setting according to your application requirements.
5	IFGainGrade	SWP/IQS/ DET/RTA	The system allows users to adjust IF gain to optimize spurious performance, linearity, and noise levels. Higher gain settings (indexed numerically) provide greater IF amplification, typically in 1 dB to 3dB increments per step. Total system gain = RF gain + IF gain. When maintaining a constant reference level (fixed total gain): 1) Increasing IF gain -> Reduces mixer input power -> Imporves spurious suppression and linearity -> Degrades noise performance; 2) Decreasing IF gain -> Increases mixer input power -> Worsens spurious/linearity -> Improves noise performance. Special cases: 1) At maximum RF gain(e.g., ref. level = -60 dBm): Further IF gain increase boosts total gain, potentially improving noise performance; 2) Below maximum RF gain(e.g., ref. level = 0 dBm): Higher IF -> Better spurious/linearity, worse noise; LowerIF gain -> Worse spurious/linearity, better noise.

6.3 Frequency

Table 5 Table of Frequency Parameters and Related Concept Descriptions

No.	Parameters	Applicable	Description
1	FreqAssignment	SWP	In SWP mode, this variable allows users to define the frequency scan range either in StartStop or CenterSpan format.
2	StartFreq_Hz StopFreq_Hz CenterFreq_Hz Span_Hz		
3	TracePointStrategy	SWP	In SWP mode, the spectrum analysis method is determined by TracePointStrategy: 1) When TracePointStrategy = BinSizeAssined: The system uses sweep-based analysis, where the trace point count is explicitly defined by TracePoints. In this mode, TraceAlign settings are ignored; 2) For other TracePointStrategy values: The system employs FFT-based analysis. Due to underlying software implementation and trace detection mechanisms, the native ananlysis frequencies cannot be perfectly aligned with the configured start/stop frequencies. The returned trace data points will slightly exceed the specified. User-adjustable handling methods: 1) Truncate endpoint data to match the desired frequency range; 2) Set TraceAlign = AlignToStart to force alignment at the start frequency (end frequency still requires truncation). Note for FFT mode: While users can specify a desired trace point count (TracePoints), hardware limitations typically prevent exact matches. The system returns the closest achievable point count.
4	TracePoints		
5	TraceAlign		

6.4 Analysis

Table 6 Table of Analysis Parameters and Related Concept Descriptions

No.	Parameters	Applicable	Description
1	SpurRejection	SWP	The system provides three spurious suppression modes: Bypass, Standard, and Enhanced. This feature effectively suppresses most composite spurious components but does not improve system residual responses. It also reduces sweep speed and time-varying signal measurement capability. 1) For steady-state signals (e.g., CW tones): Enabling spurious suppression significantly improves spurious-free dynamic range; 2) For fast time-varying signals (e.g., modulated signals): Activation may cause intermittent signal loss or inaccurate power measurements. Use with caution. Recommendation: Toggle the feature while observing spectral changes to determine if spurious rejection is suitable for your test scenario.
2	PowerBalance	SWP	In SWP mode, users can balance scan speed and power consumption by configuring the Power Consumption Balance parameter: 1) Power Consumption Blance = 0: The system operates at maximum sweep speed (highest power draw); 2) Power Consumption Balance = 40-1000 (typical range): Higher values reduce scan speed and lower power consumption. Critical Note: Higher Power Consumption Balance values significantly degrade time-varying signal detection capability. Use caution when testing highly dynamic signals.
3	Window	SWP/RTA	When performing FFT-based spectrum analysis, the system provides multiple window functions, each with distinct advantages. Please select the appropriate window based on your testing requirements: 1) FlatTop Window: Provides excellent amplitude accuracy; Significantly reduces amplitude errors caused by the picket-fence effect; Ideal for high-precision amplitude measurements; 2) Blackman-Nuttall Window: Features a narrow main lobe for high frequency resolution; Enables faster scanning speeds than FlatTop at the same RBW setting; Suitable for high-frequency-resolution and fast-sweep testing scenarios;

			3) LowSideLobe Window: Features extremely low sidelobe levels, effectively suppressing interference from strong signals to adjacent frequencies. Ideal for test scenarios requiring high dynamic range or coexistence of strong and weak signals.
4	FFTExcutionStrategy	SWP	In standard spectrum analysis mode, users can select the signal processing method: 1) Auto mode: The system automatically selects FPGA or CPU processing based on RBW; 2) FPGA-Only: Significantly reduces CPU load; Slower sweep speeds ata RBW ≤ 5KHz due to FFT size limitations; 3) CPU-Only: Supports FFT sizes > 64K points, enabling faster sweeps ata narrow RBW (≤5KHz).
5	SweepTime	SWP/RTA	In this system, the sweep time is defined as the total time required to complete one full sweep from the start frequency to the stop frequency. When SweepTime = SWTMode_Manual, this parameter represents the absolute time; when *N is specified, this parameter is the sweep time multiplier, i.e., sweep is performed at N times the minimum sweep time.
6	DecimateFactor	IQS/DET/RTA	In IQS/DET/RTA mode, the system uses DecimateFactor to realize variable analysis bandwidth. Analysis bandwidth is equal to be Analysis bandwidth (DecimateFactor = 1) / DecimateFactor. Due to the limitations, the system will achieve the nearest available value for the DecimateFactor according to the desired DecimateFactor for configuration and feedback to the user.
7	BusTimeOut	IQS/DET/RTA	BusTimeOut sets an upper limit of execution time for functions related to fetching data, and the process will be ended if valid data cannot be fetched within that time so that the system does not wait indefinitely. In IQS/DET/RTA mode, this parameter needs to be set.

6.5 Detector and Trace Detector

Detector: Under the same local frequency point, the detector ratio frame data is collected, and according to the characteristics of the detector, the multi-frame data is detected frequency by frequency point, and the eigenvalue frame is finally generated. The following figure takes PosPeak Detector as an example to introduce the process of positive peak detection, where Frame 0, Frame 1 and Frame 2 are the data collected at different moments, and After PosPeak Detector is the data after positive peak detection.

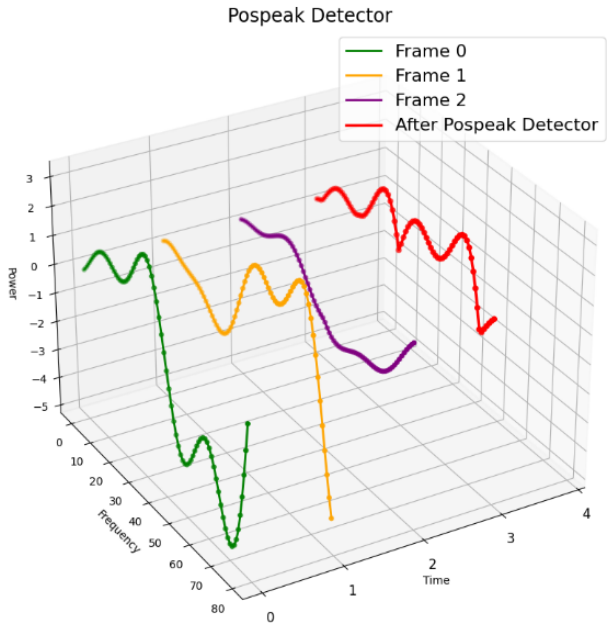


Figure 9 Positive Peak Detector

Trace Detector: According to the selected trace detector, the entire spectrum trace is detected in steps of trace detection ratio to generate the eigenvalue trace. The following figure takes PosPeak TraceDetector as an example to introduce the process of PosPeak Trace Detector, where Before PosPeak Trace Detector is the data before PosPeak Trace Detector and After PosPeak Trace Detector is the data after PosPeak Trace Detector.

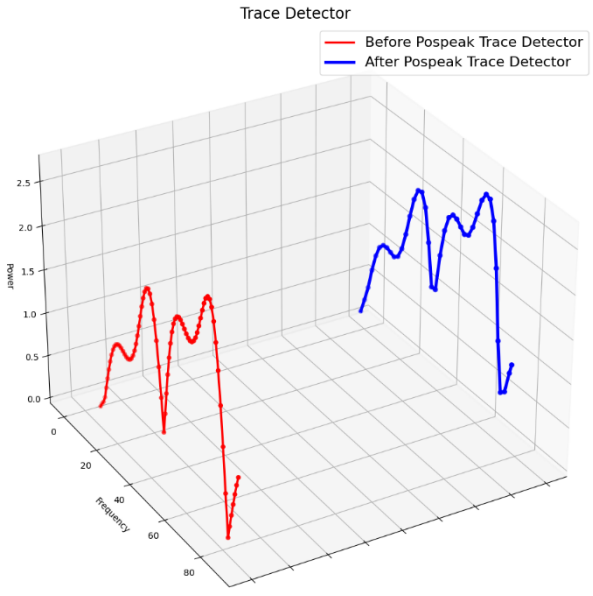


Figure 10 Positive Peak Trace Detector

6.6 Default Units

Table 7 Summary of Main Variables and Units

Variables	Unit
Frequency	Hz
Power	dBm
Voltage	V
Time	s

7. Device and System Main Functions

Before calling any hardware-related API functions, Device_Open() must first be invoked to initialize the device. Once the business logic or application tasks are completed, Device_Close() must be called to shut down the device and release allocated memory resources.

7.1 Device_List

int Device_List (**const BootProfile_TypeDef* BootProfile,** **uint8_t* Devicecount,** **uint8_t DevNum[MAX_DEVICE],** **DeviceInfo_TypeDef DeviceInfo_O[MAX_DEVICE]**)	
Description	
Query all USB devices currently connected to the host computer, along with their corresponding device IDs and device information.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **BootProfile**	Device startup configuration specifying the interface type as USB. Refer to the definition of the BootProfile_TypeDef structure for details.
[out] **Devicecount**	Returns the number of devices currently connected to the host, with a maximum of 256 devices per bus.
[out] **DevNum[MAX_DEVICE]**	Returns the device IDs of all devices currently connected to the host.
[out] **DeviceInfo_O[MAX_DEVICE]**	Returns detailed information for all connected devices, including serial number, type, MCU version, and FPGA version. Refer to the definition of the DeviceInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	None.
Example	

int Status = 0; int DevNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
uint8_t Devicecount = 0;	

```
DeviceInfo_TypeDef DeviceInfo_O[MAX_DEVICE];
vector<uint8_t>DevNumList(MAX_DEVICE);
Status = Device_List(&BootProfile, &Devicecount, DevNumList.data(), DeviceInfo_O);
for (int i = 0; i < Devicecount; i++) {
    if (DeviceInfo_O[i].Model == 57) {
        DevNum = DevNumList[i];
        break;
    }
}
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
```

7.2 Get_APIVersion

int Get_APIVersion(void)	
Description	
Retrieve the version number of the currently used API library.	
Compatibility	0.55.77 and later.
Return value	The current API version, represented by a major version, minor version, and revision (patch) version. bit[31..16] indicates the major version, bit[15..8] indicates the minor version, and bit[7..0] indicates the revision version.
Calling constraints	None.
Example	

```
int apiVersion = Get_APIVersion();
int major = 0, minor = 0, rev = 0;
major = (apiVersion >> 16) & 0xffff;
minor = (apiVersion >> 8) & 0xff;
rev = apiVersion & 0xff;
cout << "htra_api verion: " << major << "." << minor << "." << rev << endl;
```

7.3 APISupportFirmwareVersions

```
int APISupportFirmwareVersions (
        DeviceFirmwareVersion_TypeDef** Versions,
        uint32_t* Count
)
```

Description	
Get the firmware versions supported by the current API.	
Compatibility	0.55.79 and later.

Parameter description	
[out] Versions	Returns a pointer to the structure containing firmware version information supported by the current API. Refer to the definition of the DeviceFirmwareVersion_TypeDef structure for details.
[out] Count	Outputs the number of firmware version entries supported by the current API.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	None.
Example	

```
DeviceFirmwareVersion_TypeDef* pVersions = nullptr;
uint32_t count = 0;
int status = APISupportFirmwareVersions(&pVersions, &count);
if (status == 0) {
  for (uint32_t i = 0; i < count; ++i) {
    cout << "FPGA: " << pVersions[i].FFWVersion << " | MCU: " << pVersions[i].MFWVersion << " | Bus: "
    << pVersions[i].BusVersion << " | PMU: " << pVersions[i].PMUVersion << " | AGU: " <<
    pVersions[i].AGUVersion << std::endl;
  }
}
```

7.4 Device_ReEnumerate

int Device_ReEnumerate (void Device)**	
Description	
When the device is powered on and connected to a USB 3.0 port, it may be initially recognized as USB 2.0. It has been confirmed that re-plugging allows it to be correctly recognized as USB 3.0. Under this condition, calling this function enables re-enumeration from USB 2.0 to USB 3.0 at the software level, without the need for physical reconnection. After this function succeeds, it is necessary to call *Device_Close* and then *Device_Open* to restore the connection with the device.	
Compatibility	0.55.82 and later.
Parameter description	
[in] Device	Device handle
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
while (1)
{
```

```
    int Status = 0; void* Device = NULL; int DevNum = 0;
    BootProfile_TypeDef BootProfile;
    BootInfo_TypeDef BootInfo;
    BootProfile.DevicePowerSupply = USBPortAndPowerPort;
    BootProfile.PhysicalInterface = USB;
    Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
    if (BootInfo.BusSpeed == 2) {
        Device_ReEnumerate(&Device);
        Device_Close(&Device);
        std::this_thread::sleep_for(std::chrono::seconds(20));
    }
else {
        break;
    }
}
```

7.5 Device_Open

```
int Device_Open(
    void** Device,
    int DeviceNum,
    const BootProfile_TypeDef* BootProfile,
    BootInfo_TypeDef* BootInfo
)
```

Description	
Open the specified device and obtains the device handle, which is used to identify the target device in subsequent API calls. When multiple devices are present in the system, each device can be opened separately by specifying a different device index	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle, used to identify the target device in subsequent API calls.
[in] DeviceNum	Specifies the device index. When multiple devices are present, this index can be used to select the target device. The index starts from 0 and increments sequentially.
[in] BootProfile	Device boot configuration, including interface type, power supply method, and network parameters, used to initialize the device. Refer to the definition of the BootProfile_TypeDef structure for details.

[out] **BootInfo**	Return device boot information, including device details, bus speed and version, API version, as well as errors and warnings generated during the boot process. Refer to the definition of the BootInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Device_Open must be called once before invoking any other device-related functions to acquire device resources and obtain the device handle. Within the same module, it is only necessary to call this function once. Subsequent operations can be performed using the returned device handle. After completing all operations, Device_Close must be called to release the device resources
Example	Please refer to the relevant example of the Device_QueryDeviceInfo_Realtime() function.

7.6 Device_Close

int Device_Close(void Device)**	
Description	
Close the specified device and releases the resources allocated by Device_Open. This function should be called after completing all device operations.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function only needs to be called when the program is about to terminate. After calling it, the USB device connection will be closed and the allocated memory will be released. If the device needs to be used again, Device_Open must be called to re-establish the connection and reopen the device
Example	Please refer to the relevant example of the Device_QueryDeviceInfo_Realtime() function.

7.7 Device_QueryDeviceState/Device_QueryDeviceState_Realtime

int Device_QueryDeviceState(**void** Device,** **DeviceState_TypeDef* DeviceState**)	
int Device_QueryDeviceState_Realtime(**void** Device,** **DeviceState_TypeDef* DeviceState**)	
Description	
Retrieve the spectrum analyzer device status, including device temperature, hardware operating status, and geographic time information (optional feature required). Device_QueryDeviceState: Non-real-time mode. Does not interrupt data acquisition, but the information is only updated after a data packet is received; Device_QueryDeviceState_Realtime: Real-time mode. Temporarily occupies the data channel for a short period of time.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] DeviceState	Pointer to a structure containing the device status information. After the function call, this structure is updated with the latest device status. Refer to the definition of the DeviceState_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the Device_QueryDeviceInfo_Realtime() function.

7.8 Device_QueryDeviceInfo/Device_QueryDeviceInfo_Realtime

int Device_QueryDeviceInfo(**void** Device,** **DeviceInfo_TypeDef* DeviceInfo**)	
int Device_QueryDeviceInfo_Realtime(**void** Device,** **DeviceInfo_TypeDef* DeviceInfo**)	
Description	
Retrieve device information, including the device serial number, hardware version, firmware version, and other related details. Device_QueryDeviceInfo: Non-real-time mode. Does not interrupt data acquisition, but the information is only updated after a data packet is received. Device_QueryDeviceInfo_Realtime: Real-time mode. Temporarily occupies the data channel for a short.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] DeviceInfo	Structure containing the basic device information, including the device unique identifier and various version details. After the function call, this structure is updated with the latest device information. Refer to the definition of the DeviceInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); DeviceState_TypeDef DeviceState; Status = Device_QueryDeviceState(&Device, &DeviceState); Status = Device_QueryDeviceState_Realtime(&Device, &DeviceState); DeviceInfo_TypeDef DeviceInfo; Status = Device_QueryDeviceInfo(&Device, &DeviceInfo); Status = Device_QueryDeviceInfo_Realtime(&Device, &DeviceInfo); Status = Device_Close(&Device);	

7.9 Device_QueryPowerSupplyState

int Device_QueryPowerSupplyState(void** Device, PowerSupplyState_TypeDef* PowerSupplyState)	
Description	
Retrieve the device power status, including the voltage and current of the RF board power port (POWER) and the digital board USB port (DATA).	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **PowerSupplyState**	Returns the voltage and current of the device power port and data port. Refer to the definition of the PowerSupplyState_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); PowerSupplyState_TypeDef PowerSupplyState; Status = Device_QueryPowerSupplyState(&Device, &PowerSupplyState); Status = Device_Close(&Device);	

8. Device and System Other Functions

8.1 Device_SetSysPowerState

int Device_SetSysPowerState(**void** Device,** **SysPowerState_TypeDef SysPowerState**)	
Description	
Sets the system power state of the device, allowing control over the power-on or power-off of various functional areas, as well as putting the RF module into standby. This enables management of different power consumption and wake-up requirements.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] SysPowerState	Sets the system power state of the device. Refer to the definition of the SysPowerState_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); SysPowerState_TypeDef SysPowerMode = PowerON; Status = Device_SetSysPowerState(&Device, SysPowerMode); Status = Device_Close(&Device);	

8.2 Device_SetMcuSysPowerState

int Device_SetMcuSysPowerState (void** Device, SysPowerMode_TypeDef SysPowerState)	
Description	
Sets the MCU low power state of the device, allowing control of MCU power-on, standby, and power-off modes to meet different power consumption requirements.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **SysPowerState**	Set the MCU low power mode of the device. Refer to the definition of the SysPowerMode_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); SysPowerState_TypeDef SysPowerMode = SysPowerMode_Normal; Status = Device_SetSysPowerState(&Device, SysPowerMode); Status = Device_Close(&Device);	

8.3 Device_CalibrateRefClock

```
int Device_CalibrateRefClock(
    void** Device,
    ClkCalibrationSource_TypeDef ClkCalibrationSource,
    const double TriggerPeriod_s,
    const uint64_t TriggerCount,
    const bool RewriteRFCal,
    double* RefCLKFreq_Hz
)
```

Description	
Calibrates the device reference clock frequency using an external trigger signal with a known period. The function calculates the calibrated frequency based on the specified trigger period and number of triggers, and optionally allows writing the calibration result to the calibration file to serve as the default reference frequency for subsequent device startups.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **ClkCalibrationSource**	Specifies the clock calibration source. Refer to the definition of the ClkCalibrationSource_TypeDef enumeration structure for details.
[in] **TriggerPeriod_s**	The period of the known calibration signal (unit: seconds). The accuracy of this value directly affects the precision of the reference clock calibration.
[in] **TriggerCount**	The number of triggers used for calibration. For scenarios using GNSS 1PPS calibration, a higher number of triggers helps suppress jitter errors and improves calibration accuracy, but also increases the calibration time. It is recommended to use more than 30 triggers (i.e., a calibration time exceeding 30 seconds) when using GNSS 1PPS.
[in] **RewriteRFCal**	Specifies whether to write the calibration result to the calibration file. 0: Do not write. The calibration result will be lost when the device is powered off. 1: Write. The calibration result will persist even after the device is powered off and on.
[out] **RefCLKFreq_Hz**	Returns the reference clock frequency obtained from this calibration (unit: Hz).
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.

Example

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
ClkCalibrationSource_TypeDef ClkCalibrationSource = CalibrateByExternal;
double TriggerPeriod_s = 1; uint64_t CalibrationTimes = 1 * 60;
bool RewriteRFCal = false; double RefCLKFreq_Hz = 0;
Status = Device_CalibrateRefClock(&Device, ClkCalibrationSource, TriggerPeriod_s, CalibrationTimes, RewriteRFCal, &RefCLKFreq_Hz);
Status = Device_Close(&Device);

8.4 Device_SetFanState

int Device_SetFanState(**void** Device,** **const FanState_TypeDef FanState,** **const float ThreshouldTemperature**)	
Description	
Sets the operating mode of the device fan and controls its on/off state based on the specified temperature thresholds, enabling thermal management of the device. (Supported only for USB devices below 8.5GHz.	
Compatibility	0.55.0 and later.
Parameter description	
[in] Device	Device handle.
[in] FanState	Sets the operating mode of the device fan. Refer to the definition of the FanState_TypeDef enumeration structure for details.
[in] ThreshouldTemperature	Threshold temperature (in degrees Celsius). When FanState = FAN_AUTO, the system turns on the fan if the device temperature exceeds this threshold, and turns it off when the temperature drops 10°C below the threshold.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL;	

```
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
FanState_TypeDef FanState = FanState_On;
float Temperature = 0;
Status = Device_SetFanState(&Device, FanState, Temperature);
Status = Device_Close(&Device);
```

8.5 Devcie_SetFreqResponseCompensation

```
int Devcie_SetFreqResponseCompensation(
    void** Device,
    uint8_t State,
    const double *Frequency_Hz,
    const float *CorrectVal_dB,
    uint8_t Points
)
```

Description

In SWP mode, a frequency compensation mechanism is used to correct the power for the specified frequency band. The compensation rules are as follows:

1. Interpolation within the frequency compensation range:

Linear interpolation is applied between adjacent frequency points in the Frequency_Hz array, ensuring that the power compensation values transition smoothly within the specified frequency range

2. From the start frequency to the first frequency in the compensation array (Frequency_Hz[0]):

The compensation value for this range is CorrectVal_dB[0], i.e., the first value in the compensation array is used to fill this range.

3. From the last frequency in the compensation array (Frequency_Hz[Points - 1]) to the end frequency:

The compensation value for this range is CorrectVal_dB[Points - 1], i.e., the last value in the compensation array is used to fill this range.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **State**	Specifies whether to enable frequency response compensation. 0: Disable compensation. 1: Enable compensation.
[in] **Frequency_Hz**	Array of frequency compensation values.
[in] **CorrectVal_dB**	Array of power compensation values.

[in] **Points**	Number of compensation points, with a maximum of 256 points.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.StartFreq_Hz = 2e8;
SWP_ProfileIn.StopFreq_Hz = 3e8;
Status = SWP_Configuration(&Device, &SWP_ProfileIn, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
uint8_t compensation_points = 4;
vector<double> Compensate_freq = { 239.999e6, 240e6, 260e6, 260.001e6 };
vector<float> Compensate_dBm = { 0.0f, 10.0f, 30.0f, 0.0f };
Status = Devcie_SetFreqResponseCompensation(&Device, 1, Compensate_freq.data(),
Compensate_dBm.data(), 4);
MeasAuxInfo_TypeDef MeasAuxInfo;
while (1) {
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
}
Status = Device_Close(&Device);
```

8.6 Devcie_SetFreqResponseCompensation_PM1

```
int Devcie_SetFreqResponseCompensation_PM1(
    void** Device,
    uint8_t State,
    const double *Frequency_Hz,
    const float *CorrectVal_dB,
    uint32_t Points
)
```

Description

In SWP mode, a frequency compensation mechanism is used to perform power correction for the specified frequency band. For the compensation rules, refer to the relevant description of the Devcie_SetFreqResponseCompensationfunction.	
Compatibility	0.55.84 and later.
Parameter description	
[in] Device	Device handle.
[in] State	Enable or disable frequency response compe 0: Compensation disabled1: Compensation enabled.
[in] Frequency_Hz	Frequency compensation array.
[in] CorrectVal_dB	Power compensation data array.
[in] Points	Number of compensation points , with a maximum of 65,536 points.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function must be called after SWP_Configuration.
Example	Refer to the relevant examples of the Devcie_SetFreqResponseCompensationfunction.

8.7 Device_SetGNSSDataSource

int Device_SetGNSSDataSource (**void** Device,** **GNSSDataSource_TypeDef *GNSSDataSourceIn** **GNSSDataSource_TypeDef *GNSSDataSourceOut** **)**
Description
Set GNSS date source.

Compatibility	0.55.84 and later.
Parameter description	
[in] Device	Device handle.
[in] GNSSDataSourceIn	Refer to the definition of the GNSSDataSource_TypeDefstructure for details.
[in] GNSSDataSourceOut	Refer to the definition of the GNSSDataSource_TypeDefstructure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function must be called after SWP_Configuration.
Example	Refer to the relevant examples of the Devcie_SetFreqResponseCompensationfunction.

8.8 Device_GetNetworkDeviceList

int Device_GetNetworkDeviceList(uint8_t*DeviceCount, NetworkDeviceInfo_TypeDef DeviceInfo[64], uint8_t LocalIP[4], uint8_t LocalMask[4])	
Description	
When using an Ethernet-type device, retrieves the IP addresses and subnet masks of all devices on the network, and also returns the IP address and subnet mask of the local network interface.	
Compatibility	0.55.77 and later.
Parameter description	
[out] DeviceCount	Return the number of devices detected on the network.
[out] DeviceInfo[64]	Returns information for each device detected on the network, including the device serial number, device type, hardware version, MCU and FPGA firmware versions, as well as the IP address and subnet mask. Refer to the definition of the NetworkDeviceInfo_TypeDef structure for details.
[out] LocalIP[4]	Output the local IP address.
[out] LocalMask[4]	Output the local subnet mask.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	None.
Example	
int Status = -1; uint8_t DeviceCount = 0; uint8_t LocalIP[4]; uint8_t LocalMask[4]; NetworkDeviceInfo_TypeDef DeviceInfo[64]; Status = Device_GetNetworkDeviceList(&DeviceCount, DeviceInfo, LocalIP, LocalMask);	

8.9 Device_SetNetworkDeviceIP

int Device_SetNetworkDeviceIP(const uint64_t DeviceUID, const uint8_t IPAddress[4], const uint8_t SubnetMask[4])	
Description	
When using an Ethernet-type device, configures the network device's IP address and subnet mask based on the device's unique serial number, enabling network parameter setup and device communication management.	
Compatibility	0.55.77 and later.

Parameter description	
[in] **DeviceUID**	Specifies the serial number of the device.
[in] **IPAddress[4]**	Specifies the IP address to be configured.
[in] **SubnetMask[4]**	Specifies the subnet mask to be configured.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; uint64_t DeviceUID = 31325119004c0048; uint8_t IPAddress[4] = { 192, 168, 2, 100}; uint8_t SubnetMask[4] = {255, 255, 255, 0}; Status = Device_SetNetworkDeviceIP(DeviceUID, IPAddress, SubnetMask);	

8.10 Device_SetNetworkDeviceIP_PM1

int Device_SetNetworkDeviceIP_PM1(**const uint8_t DeviceIP[4],** **const uint8_t IPAddress[4],** **const uint8_t SubnetMask[4]**)	
Description	
When using an Ethernet-type device, configures a new IP address and subnet mask for the device by specifying its current IP address, enabling network parameter updates and device communication management.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DeviceIP[4]**	Specify the device's current IP address, used to locate the device.
[in] **IPAddress[4]**	Specify the new IP address to be configured.
[in] **SubnetMask[4]**	Specify the new subnet mask to be configured.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	None.
Example	
int Status = -1; uint8_t DeviceIP[4] = {192, 168, 1, 100}; uint8_t IPAddress[4] = { 192, 168, 2, 100}; uint8_t SubnetMask[4] = {255, 255, 255, 0}; Status = Device_SetNetworkDeviceIP_PM1(DeviceIP, IPAddress, SubnetMask);	

8.11 Device_GetFullUID

int Device_GetFullUID(void** Device, uint64_t* UID_L64, uint32_t* UID_H32)	
Description	
Retrieves the complete UID information of the device, including the high 32 bits and low 64 bits. This is done in non-real-time mode, so it does not interrupt data acquisition, and the information is updated after a data packet is received.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **UID_L64**	Output the lower 64 bits of the device UID.
[out] **UID_H32**	Output the upper 32 bits of the device UID.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); uint64_t UID_L64; uint32_t UID_H32; Status = Device_GetFullUID(&Device, & UID_L64, & UID_H32); Status = Device_Close(&Device);	

8.12 Device_GetHardwareState

int Device_GetHardwareState(void** Device, HardWareState_TypeDef* HardWareState)
Description
Retrieve the device hardware status information, including GNSS peripheral type, receiver type, OCXO type, as well as the support status of internal clock, signal source, ADC variable sampling rate, and intermediate frequency (IF) filters.

Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] HardWareState	Output the device hardware status information, including GNSS peripheral type, signal source support, ADC variable sampling rate, and other functional states. Refer to the definition of the HardWareState_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
HardWareState_TypeDef HardWareState;
Status = Device_GetHardwareState (&Device, & HardWareState);
Status = Device_Close(&Device);
```

8.13 Device_QueryDeviceInfoWithBus

```
int Device_QueryDeviceInfoWithBus(
    int DeviceNum,
    const BootProfile_TypeDef* BootProfile,
    BootInfo_TypeDef* BootInfo
)
```

Description	
Query device information by device index and returns the basic device information, bus speed and version, API version, as well as any errors and warnings encountered during the startup process.	
Compatibility	0.55.77 and later.
Parameter description	
[in] DeviceNum	Specify the device index to be opened.
[in] BootProfile	Device startup configuration, including interface type, power supply method, and network parameters, used to initialize the device. Refer to the definition of the BootProfile_TypeDef structure for details.

[out] **BootInfo**	Return the device startup information, including device details, bus speed and version, API version, and any errors or warnings encountered during the startup process. Refer to the definition of the BootInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before Device_Open.
Example	
int Status = -1; int DeviceNum = 0; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_QueryDeviceInfoWithBus(DeviceNum, &BootProfile, &BootInfo);	

8.14 Device_SetFreqScan

int Device_SetFreqScan(**void** Device,** **double StartFreq_Hz,** **double StopFreq_Hz,** **uint16_t SweepPts** **)**
Description
Configure the device frequency sweep parameters, including start frequency, stop frequency, and number of sweep points, for spectrum scanning setup.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **StartFreq_Hz**	Start frequency of the sweep, in Hz.
[in] **StopFreq_Hz**	Stop frequency of the sweep, in Hz.
[in] **SweepPts**	The total number of frequency points to be swept.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo;	

```
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
double StartFreq_Hz = 1e9; double StopFreq_Hz = 2e9;
uint16_t SweepPts = 5;
Status = Device_SetFreqScan(&Device, StartFreq_Hz, StopFreq_Hz, SweepPts);
Status = Device_Close(&Device);
```

8.15 Device_GetFreqPlanning

```
void Device_GetFreqPlanning(
    void** Device,
    double rf,
    uint8_t* rfb,
    uint8_t* convert,
    double* if1,
    double* im,
    double* rflo,
    double* iflo,
    double* if2
)
```

Description

Calculate and return the internal RF front-end frequency planning scheme based on the input RF target frequency, including the RF band, frequency conversion strategy, first and second intermediate frequencies, and the corresponding local oscillator frequencies.

Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **rf**	Input target RF frequency, in Hz.
[out] **rfb**	RF band where the target RF frequency is located.
[out] **convert**	Frequency conversion strategy: up-conversion or down-conversion.
[out] **if1**	Return the first intermediate frequency (IF1), in Hz.
[out] **im**	Return the image frequency, in Hz.
[out] **rflo**	Return the RF local oscillator frequency, in Hz.
[out] **iflo**	Return the IF local oscillator frequency, in Hz.
[out] **if2**	Return the second intermediate frequency (IF2), in Hz.
Return value	None.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
```

```
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
double rf = 1e9;
uint8_t rfb = 0, convert = 0;
double if1 = 0, im = 0, rflo = 0, iflo = 0, if2 = 0;
Device_GetFreqPlanning(&Device, rf, &rfb, &convert, &if1, &im, &rflo, &iflo, &if2);
Status = Device_Close(&Device);
```

8.16 Device_SetIFOutput

```
int Device_SetIFOutput(
    void** Device,
    uint8_t* state,
)
```

Description	
IF output control.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **state**	Control the IF output. 0: Disable IF output; 1: Enable IF output.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
uint8_t state = 1;
Status = Device_SetIFOutput(&Device, state);
Status = Device_Close(&Device);
```

8.17 Device_QueryVersion_PMU

int Device_QueryVersion_PMU(void** Device, uint32_t* Version)	
Description	
Query the version number of the device Power Management Unit (PMU).	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **Version**	Current PMU version, represented by major version, minor version, and revision version. bit[31..16] indicates the major version, bit[15..8] indicates the minor version, and bit[7..0] indicates the revision version.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
uint32_t PMUVersion = 0;
Status = Device_QueryVersion_PMU(&Device, &PMUVersion);
Status = Device_Close(&Device);	

8.18 Device_QueryVersion_AGU

int Device_QueryVersion_AGU(void** Device, uint32_t* Version)
Description
Query the version number of the AGU.

Compatibility	0.55.79 and later.
Parameter description	
[in] Device	Device handle.
[out] Version	Current AGU version, represented by major version, minor version, and revision version. bit[31..16] indicates the major version, bit[15..8] indicates the minor version, and bit[7..0] indicates the revision version.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
uint32_t AGUVersion = 0;
Status = Device_QueryVersion_AGU(&Device, &AGUVersion);
Status = Device_Close(&Device);	

8.19 Device_InitIFAGC

int Device_InitIFAGC(void Device)**	
Description	
Initialize IF AGC and reset the AGC parameters to their default values.	
Compatibility	0.55.79 and later.
Parameter description	
[in] Device	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the Device_SetIFAGCTarget() function.

8.20 Device_SetIFAGCTarget

int Device_SetIFAGCTarget(void** Device, double* Target)	
Description	
Set the target power of the IF AGC.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in/out] **Target**	Set the target power level as the dBFs value relative to ADC full-scale. Range: 0 to -30 dBFs.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
Status = Device_InitIFAGC(&Device);
double Target = 10;
Status = Device_SetIFAGCTarget(&Device, &Target);
Status = Device_Close(&Device);	

8.21 Device_SetIFAGCPeriod

int Device_SetIFAGCPeriod(void**Device double*Period)	
Description	
Sets the AGC period and adjusts the AGC operation logic based on the specified period.	
Compatibility	0.55.79 and later.
Parameter description	

[in] **Device**	Device handle.
[in] **Period**	Sets the execution period of AGC. Range: -1 s to +2,147,483 s. When Period < 0 , AGC is executed once before sampling and is not executed during sampling. When Period = 0 , dynamic AGC is enabled. When Period > 0 , AGC is executed periodically according to the specified interval.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function must be called after Device_Open.

8.22 Device_ConvertEpochToReadable

```
void Device_ConvertEpochToReadable(
    uint64_t nsSinceEpoch,
    int16_t* year,
    int16_t* mon,
    int16_t* day,
    int16_t* hour,
    int16_t* min,
    int16_t* sec,
    int16_t* ms,
    int16_t* us,
    int16_t* ns
)
```

Description	
Convert the nanosecond-level epoch timestamp (nsSinceEpoch) to physical time (UTC).	
Compatibility	0.55.77 and later.
Parameter description	
[in] **nsSinceEpoch**	Input the nanosecond-level timestamp (ns) corresponding to the current data packet.
[out] **year**	Return the year.
[out] **mon**	Return the month.
[out] **day**	Return the day.
[out] **hour**	Return the hour.
[out] **min**	Return the minute.
[out] **sec**	Return the second.
[out] **ms**	Return the millisecond.

[out] **us**	Return the microsecond.
[out] **ns**	Return the nanosecond.
Return value	None.
Calling constraints	This function must be called after GNSS lock is achieved and valid data has been obtained by calling the Get function.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
GNSSAntennaState_TypeDef GNSSAntennaState = GNSS_AntennaExternal;
Status = Device_SetGNSSAntennaState(&Device, GNSSAntennaState);
GNSSInfo_TypeDef GNSSInfo = { 0 };
while (!GNSSInfo.GNSS_LockState) {
    Status = Device_GetGNSSInfo_Realtime(&Device, &GNSSInfo);
    this_thread::sleep_for(chrono::milliseconds(1000));
}
SWP_Profile_TypeDef SWP_ProfileIn, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
Status = SWP_Configuration(&Device, &SWP_ProfileIn, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.PartialsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.PartialsweepTracePoints);
int HopIndex = 0;
int FrameIndex = 0;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetPartialSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &HopIndex, &FrameIndex, &MeasAuxInfo);
int16_t year = 0, mon = 0, day = 0, hour = 0, min = 0, sec = 0, ms = 0, us = 0, ns = 0;
Device_ConvertEpochToReadable(MeasAuxInfo.nsSinceEpoch, &year, &mon, &day, &hour, &min, &sec, &ms, &us, &ns);
Status = Device_Close(&Device);
```

8.23 Device_GetAmpAttenState

void Device_GetAmpAttenState(**void** Device,** **PreamplifierState_TypeDef* amp,** **int8_t* att,** **uint8_t* sp**)	
Description	
Query the gain mapping parameters.	
Compatibility	0.55.79 and later.
Parameter description	
[in] Device	Device handle.
[out] amp	Return the preamplifier state. Refer to the definition of the PreamplifierState_TypeDef structure for details.
[out] att	Return the spectrum analyzer channel attenuation, in dB.
[out] sp	Return the gain space value, an internal parameter that does not require user attention or usage.
Return value	None.
Calling constraints	Must be called after xxx_Configuration related-function.
Example	

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn;
SWP_Profile_TypeDef SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.Atten = 3;
Status = SWP_Configuration(&Device, &SWP_ProfileIn, &SWP_ProfileOut, &TraceInfo);
PreamplifierState_TypeDef ampState; int8_t att; uint8_t sp;
Device_GetAmpAttenState(&Device, &ampState, &att, &sp);
Status = Device_Close(&Device);
```

8.24 Device_QueryEIO_Version_UID

int Device_QueryEIO_Version_UID(**void** Device,** **uint16_t* EIOVersion,** **uint64_t* EIOUID** **)**	
Description	
Retrieve the version and UID of the GNSS module connected to the specified device.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] EIOVersion	Return the GNSS module version number, represented by major and minor revisions. bit[15..8] indicates the major version, and bit[7..0] indicates the minor revision.
[out] EIOUID	Return the UID of the GNSS module.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = 0; int DevNum = 0; void* Device = NULL;
uint16_t EIOVersion = 0;
uint64_t EIOUID = 0;
int major = 0, minor = 0, rev = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
Status = Device_QueryEIO_Version_UID(&Device, &EIOVersion, &EIOUID);
major = (EIOVersion >> 16) & 0xffff;
minor = (EIOVersion >> 8) & 0xff;
rev = EIOVersion & 0xff;
Status = Device_Close(&Device);	

8.25 Device_GPIOSetBits

int Device_GPIOSetBits(void** Device, uint16_t Bits)	
Description	
Set the EIO option GPIO pin high.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **Bits**	GPIO control bits.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the Device_GPIOBandSwitch() function.

8.26 Device_GPIOResetBits

int Device_GPIOResetBits(void** Device, uint16_t Bits)	
Description	
Set the EIO option GPIO pin low.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **Bits**	GPIO control bits.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the Device_GPIOBandSwitch() function.

8.27 Device_GPIOBandSwitch

int Device_GPIOBandSwitch(void** Device, uint16_t Bands, double StartFreq[], double StopFreq[], uint16_t SetBits[], uint16_t ResetBits[], uint32_t delay_us[])	
Description	
EIO option GPIO pins follow the segmented frequency changes.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **Bands**	Number of frequency bands.
[in] **StartFreq[]**	Start frequency, in Hz.
[in] **StopFreq[]**	Stop frequency, in Hz.
[in] **SetBits[]**	Set GPIO pin high.
[in] **ResetBits[]**	Set GPIO pin low.
[in] **delay_us[]**	Frequency band switching delay.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = 0; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
std::cout << "Device_Open: " << Status << std::endl;
if (Status) {
    return 0;
}
// set the number of frequency bands
```

```
uint16_t Bands = 3;
// band 0: 1e9 to 1.5e9
// band 1: 3e9 to 5e9
// band 2: 7e9 to 8e9
double StartFreq[] = { 1e9,3e9,7e9 };
double StopFreq[] = { 1.5e9,5e9,8e9 };
// set GPIO
// pull up GPIO 0, 1, 2, all
// Device_GPIOSetBits(&Device, 0x01)
// Device_GPIOSetBits(&Device, 0x02)
// Device_GPIOSetBits(&Device, 0x04)
// Device_GPIOSetBits(&Device, 0xFF)
uint16_t setBits[] = { 0x01,0x02,0x04 };
// pull down GPIO 1&2
// Device_GPIOResetBits(&Device, 0x06)
uint16_t resetBits[] = { 0x06,0x05,0x03 };
uint32_t delays[] = { 100,100,100 };
Status = Device_GPIOBandSwitch(&Device, Bands, StartFreq, StopFreq, setBits, resetBits, delays);
Status = Device_Close(&Device);
```

8.28 Device_ExtMixerConfig

```
int Device_ExtMixerConfig(
    void** Device,
    ExtMixer_TypeDef* ExtMixer_In,
    ExtMixer_TypeDef* ExtMixer_Out
)
```

Description

Initializes the external mixer parameters.

Compatibility	0.55.99 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **ExtMixer_In**	Input external mixer configuration structure used to configure parameters including whether the external mixer is enabled, LO output mode, harmonic number, LO position, RF start and stop frequencies, IF frequency, and LO power. Refer to the definition of the ExtMixer_TypeDef structure for details.

[out] **ExtMixer_Out**	Output structure containing the external mixer configuration that is actually applied. Refer to the definition of the ExtMixer_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
int status = -1;
ExtMixer_TypeDef extMixerIn, extMixerOut;
extMixerIn.Enable = 1;
extMixerIn.EnableLOOut = 0;
extMixerIn.Harmonic = 18;
extMixerIn.IsHighLO = 1;
extMixerIn.StartFreq_Hz = 170e9;
extMixerIn.StopFreq_Hz = 260e9;
extMixerIn.IFFreq_Hz = 3.45e9;
extMixerIn.LOPower_dBm = 15;
Status = Device_ExtMixerConfig(device, &extMixerIn, &extMixerOut);
Status = Device_Close(&Device);
```

8.29 Device_GetDecimateFactorList

```
int Device_GetDecimateFactorList(
    void** Device,
    uint32_t* DecimateFactorList,
    uint32_t MaxCount,
    uint32_t* ActualCount
)
```

Description	
Get the list of decimation factors supported by the device.	
Compatibility	0.55.99 and later.
Parameter description	

[in] **Device**	Device handle.
[out] **DecimateFactorList**	List of decimation factors supported by the device. The array capacity should be no less than MaxCount.
[in] **MaxCount**	Maximum number of elements that DecimateFactorList can hold.
[out] **ActualCount**	Actual number of valid decimation factors returned.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = 0;void* Device = NULL;int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
const uint32_t MaxCount = 256;
vector<uint32_t> DecimateFactorList(MaxCount, 0);
uint32_t ActualCount = 0;
Status = Device_GetDecimateFactorList (&Device, DecimateFactorList.data(), MaxCount,
&ActualCount);
Status = Device_Close(&Device);
```

8.30 Device_GetOnboardMemorySize

```
int Device_GetOnboardMemorySize(
    void** Device,
    uint64_t* MemorySize
)
```

Description	
Get the onboard memory capacity of the device, in bytes.	
Compatibility	0.55.99 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **MemorySize**	Return the onboard memory capacity of the device, in bytes.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = 0;void* Device = NULL;int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
uint64_t MemorySize = 0;
Status = Device_GetOnboardMemorySize(&Device, &MemorySize);
Status = Device_Close(&Device);
```

8.31 Device_GetBusBandwidth

int Device_GetBusBandwidth (

void Device,**

uint64_t* Bandwidth

)

Description	
Get the device bus bandwidth, in bytes/s.	
Compatibility	0.55.99 and later.
Parameter description	
[in] Device	Device handle.
[out] Bandwidth	Returns the device bus bandwidth, in bytes/s.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = 0;void* Device = NULL;int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
uint64_t Bandwidth = 0;
Status = Device_GetBusBandwidth(&Device, &Bandwidth);
Status = Device_Close(&Device);
```

8.32 Device_GetSupportedFunctions

int Device_GetSupportedFunctions (void** Device, uint64_t* SupportedFunctions)	
Description	
Gets the list of functions supported by the device. Each bit in SupportedFunctions indicates whether the corresponding function is supported. For the specific bit masks, see DEVICE_FUNCTION_xxx.	
Compatibility	0.55.99 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **SupportedFunctions**	Returns the list of functions supported by the device.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = 0;void* Device = NULL;int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
uint64_t SupportedFunctions = 0;
Status = Device_GetSupportedFunctions(&Device, &SupportedFunctions);
Status = Device_Close(&Device);	

9. System Functions Related to Device and GNSS

9.1 Device_SetGNSSAntennaState

int Device_SetGNSSAntennaState(void** Device, const GNSSAntennaState_TypeDef GNSSAntennaState)	
Description	
Sets the GNSS antenna status, allowing control of the antenna's power or operating mode (optional feature required).	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **GNSSAntennaState**	Set the GNSS antenna status. Refer to the definition of the GNSSAntennaState_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); GNSSAntennaState_TypeDef GNSSAntennaState = GNSS_AntennaExternal; Status = Device_SetGNSSAntennaState(&Device, GNSSAntennaState); Status = Device_Close(&Device);	

9.2 Device_SetGNSSxpps

int Device_SetGNSSxpps(void** device, uint8_t enableout, double xpps, double delay)	
Description	
Configure the GNSS pulse per second.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **enableout**	Enable or disable the GNSS PPS (pulse per second). 1: Enable; 0: Disable.
[in] **xpps**	Set the PPS period. Range: 0.25 to 10^7.
[in] **delay**	Set the delay of the GNSS PPS. Higher accuracy is achieved after GNSS lock. Range: 0 to 1 / xpps.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
uint8_t Enableout = 1;double xpps= 1; double delay =0;
Status = Device_SetGNSSxpps (&Device, Enableout, xpps, delay);
Status = Device_Close(&Device);	

9.3 Device_SetDOCXOWorkMode

int Device_SetDOCXOWorkMode(void** Device, const DOCXOWorkMode_TypeDef DOCXOWorkMode)	
Description	
When using the GNSS functionality, sets the operating status of the DOCXO.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **DOCXOWorkMode**	Set the DOCXO operating mode. Refer to the definition of the DOCXOWorkMode_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); DOCXOWorkMode_TypeDef DOCXOWorkMode = DOCXO_LockMode; Status = Device_SetDOCXOWorkMode(&Device, DOCXOWorkMode); Status = Device_Close(&Device);	

9.4 Device_GetGNSSInfo

int Device_GetGNSSInfo(void** Device, GNSSInfo_TypeDef* GNSSInfo)	
Description	
When using the GNSS function, obtain the GNSS device status information (option required). Device_GetGNSSInfo: Non real-time mode. It does not interrupt data acquisition, but the information is only updated after a data packet is received	
Compatibility	0.55.77 later.

Parameter description	
[in] **Device**	Device handle.
[out] **GNSSInfo**	Return the GNSS device status information, including positioning coordinates, altitude, satellite count, GNSS and DOCXO status, antenna status, etc. Refer to the definition of the GNSSInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); Status = Device_SetGNSSAntennaState(&Device, GNSS_AntennaExternal); GNSSInfo_TypeDef GNSSInfo; Status = Device_GetGNSSInfo(&Device, & GNSSInfo); Status = Device_Close(&Device);	

9.5 Device_GetGNSS_SatDate/Device_GetGNSS_SatDate_Realtime

int Device_GetGNSS_SatDate(void** Device, GNSS_SatDate _TypeDef* GNSS_SatDate)
int Device_GetGNSS_SatDate_Realtime(void** Device, GNSS_SatDate_TypeDef* GNSS_SatDate)
Description
Retrieve the current GNSS satellite signal-to-noise ratio (SNR) information of the device (optional feature required), including the number of visible satellites and the signal strength of satellites used for positioning. Device_GetGNSS_SatDate: Non-real-time mode. Does not interrupt data acquisition, but the information is updated only after a data packet is received. Device_GetGNSS_SatDate_Realtime: Real-time mode. Temporarily occupies the data channel for a short period of time.

Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] GNSS_SatDate	Return the GNSS satellite information of the current device, including the number of visible satellites and the satellites used for positioning. Refer to the definition of the GNSS_SatDate_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open. Note: The signal-to-noise ratio (SNR) and satellite count are valid only after GNSS lock is achieved; otherwise, the default values are 0.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
Status = Device_SetGNSSAntennaState(&Device, GNSS_AntennaExternal);
GNSS_SatDate_TypeDef GNSS_SatDate;
Status = Device_GetGNSS_SatDate(&Device, &GNSS_SatDate);
Status = Device_GetGNSS_SatDate_Realtime(&Device, &GNSS_SatDate);
Status = Device_Close(&Device);
```

10. Standard Spectrum Analysis Main Functions

10.1 SWP_ProfileDeInit

int SWP_ProfileDeInit(void** Device, SWP_Profile_TypeDef* UserProfile_O)	
Description	
Initializes the configuration parameter set for SWP mode (SWP_Profile_TypeDef). SWP_Profile_TypeDef defines all device parameters in SWP mode, including frequency, reference level, resolution bandwidth (RBW), and other related settings.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **UserProfile_O**	Outputs the default configuration parameter set. Refer to the definition of the SWP_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the SWP_GetPartialSweep() function.

10.2 SWP_Configuration

int SWP_Configuration(void** Device, const SWP_Profile_TypeDef* ProfileIn, SWP_Profile_TypeDef* ProfileOut, SWP_TraceInfo_TypeDef* TraceInfo)	
Description	
Configures the spectrum analyzer device to operate in Standard Spectrum Analysis mode and sets the relevant parameters for this mode. In SWP mode, parameters such as frequency, reference level, resolution bandwidth, and others are encapsulated in the SWP_Profile_TypeDef structure	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.

[in] **ProfileIn**	Configuration profile input. Refer to the definition of the SWP_Profile_TypeDef structure for details.
[out] **ProfileOut**	Configuration profile output. Not all parameters in ProfileIn can be strictly applied; the device may automatically adjust certain parameters according to hardware capabilities or internal constraints. The actual applied configuration shall be based on the output profile. Refer to the definition of the SWP_Profile_TypeDef structure for details.
[out] **TraceInfo**	Returns the relevant information of the spectrum trace. Refer to the definition of the SWP_TraceInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_ProfileDeInit.
Example	Please refer to the relevant example of the SWP_GetPartialSweep() function.

10.3 SWP_AutoSet

```
int SWP_AutoSet(
    void** Device,
    SWPApplication_TypeDef Application,
    const SWP_Profile_TypeDef* ProfileIn,
    SWP_Profile_TypeDef* ProfileOut,
    SWP_TraceInfo_TypeDef* TraceInfo,
    uint8_t ifDoConfig
)
```

Description	
In Standard Spectrum Analysis (SWP) mode, provides recommended device configuration based on the application objective. In SWP mode, parameters such as frequency, reference level, resolution bandwidth (RBW), and others are encapsulated in the SWP_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **Application**	Set the measurement type. Refer to the definition of the SWPApplication_TypeDef enumeration structure for details.
[in] **ProfileIn**	Configuration profile input. Refer to the definition of the SWP_Profile_TypeDef structure for details.
[out] **ProfileOut**	Configuration profile output. Refer to the definition of the SWP_Profile_TypeDef structure for details.

[out] **TraceInfo**	Return information related to the spectrum trace. Refer to the definition of the SWP_TraceInfo_TypeDef structure for details.
[in] **ifDoConfig**	Specify whether it is necessary to call the SWP_Configuration() function separately. 0: Required; 1: Not required.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	When ifDoConfig is 0, SWP_Configuration must be called beforehand. When ifDoConfig is 1, the function internally calls SWP_Configuration, so no additional call to SWP_Configuration is required

Example: (ifDoConfig = 1)

```
int Status = -1;int DeviceNum = 0;void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef ProfileIn, ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
uint8_t ifDoConfig=1;
SWPApplication_TypeDef Application;
Status = SWP_ProfileDeInit(&Device, &ProfileIn);
Status = SWP_AutoSet (&Device, Application ,&ProfileIn, &ProfileOut, &TraceInfo, ifDoConfig);
Status = Device_Close(&Device);
```

10.4 SWP_GetPartialSweep

```
int SWP_GetPartialSweep(
    void** Device,
    double Freq_Hz[],
    float PowerSpec_dBm[],
    int* HopIndex,
    int* FrameIndex,
    MeasAuxInfo_TypeDef* MeasAuxInfo
)
```

Description

Retrieves the spectrum results at each frequency hopping point in SWP mode, while also returning the current hop index, frame index, and auxiliary information of the measurement data. This allows the user to assemble multiple scans into a complete spectrum curve. The function also returns its execution status.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **Freq_Hz[]**	Returns the frequency array corresponding to the current hop point, in Hz. The array size equals TraceInfo.PartialSweepTracePoints.
[out] **PowerSpec_dBm[]**	Returns the amplitude array corresponding to the current hop point, in dBm. The array size equals TraceInfo.PartialSweepTracePoints.
[out] **HopIndex**	Return the hop index of the data.
[out] **FrameIndex**	Returns the frame index of the data. This is valid only when the sweep time mode is not SWTMode_minSWT and the detector is set to MaxPower.
[out] **MeasAuxInfo**	Returns the auxiliary information of the data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn,SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.StartFreq_Hz = 9e3;
SWP_ProfileIn.StopFreq_Hz = 6.35e9;
SWP_ProfileIn.RBWMode = RBW_Manual;
SWP_ProfileIn.RBW_Hz = 200e3;
Status = SWP_Configuration(&Device, &SWP_ProfileIn, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
int HopIndex = 0, FrameIndex = 0;
MeasAuxInfo_TypeDef MeasAuxInfo;
```

```
for (int i = 0; i < TraceInfo.TotalHops; i++) {
Status = SWP_GetPartialSweep(&Device, Frequency.data() + i * TraceInfo.PartialsweepTracePoints, PowerSpec_dBm.data() + i * TraceInfo.PartialsweepTracePoints, &HopIndex, &FrameIndex, &MeasAuxInfo);
}
Device_Close(&Device);
```

10.5 SWP_GetFullSweep

```
int SWP_GetFullSweep(
    void** Device,
    double Freq_Hz[],
    float PowerSpec_dBm[],
    MeasAuxInfo_TypeDef* MeasAuxInfo
)
```

Description	
Retrieves the complete spectrum result in SWP mode along with the auxiliary information of the measurement data, and also returns the function execution status.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **Freq_Hz[]**	Returns the frequency array, in Hz. The array size equals TraceInfo.PartialSweepTracePoints.
[out] **PowerSpec_dBm[]**	Returns the amplitude array, in dBm. The array size equals TraceInfo.PartialSweepTracePoints.
[out] **MeasAuxInfo**	Returns the auxiliary information of the data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Configuration.
Example	

```
int Status = -1;int DeviceNum = 0;void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef ProfileIn;
SWP_Profile_TypeDef ProfileOut;
```

```
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &ProfileIn);
Status = SWP_Configuration(&Device, &ProfileIn, &ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
Status = Device_Close(&Device);
```

11. Standard Spectrum Analysis Other Functions

11.1 SWP_GetPartialSweep_PM1

int SWP_GetPartialSweep_PM1(**void** Device,** **SWPTrace_TypeDef * PartialTrace** **)**	
Description	
Polymorphic version of SWP_GetPartialSweep. Based on the original function, it modifies the format of the returned data to enhance data encapsulation.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] PartialTrace	Returns the top-level structure containing the configuration, returned information, and temporary data. Refer to the definition of the SWPTrace_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef ProfileIn;
SWP_Profile_TypeDef ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &ProfileIn);
Status = SWP_Configuration(&Device, &ProfileIn, &ProfileOut, &TraceInfo);
SWPTrace_TypeDef PartialTrace;
vector<double> Frequency(TraceInfo.PartialsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.PartialsweepTracePoints);
PartialTrace.Freq_Hz = Frequency.data();
PartialTrace.PowerSpec_dBm = PowerSpec_dBm.data();
Status = SWP_GetPartialSweep_PM1(&Device, &PartialTrace);
Status = Device_Close(&Device);
```

11.2 SWP_ResetTraceHold

int SWP_ResetTraceHold (void Device)**	
Description	
When the trace update mode (SWP_TraceType_TypeDef) is set to MaxHold or MinHold, the held trace data is reset.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	None.
Calling constraints	Effective only when SWP_TraceType_TypeDef is set to MaxHold or MinHold. Must be called after the SWP_Get related-function.
Example	

```
int Status = -1;int DeviceNum = 0;void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef ProfileIn;
SWP_Profile_TypeDef ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &ProfileIn);
Status = SWP_Configuration(&Device, &ProfileIn, &ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.PartialsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.PartialsweepTracePoints);
int HopIndex = 0;int FrameIndex = 0;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetPartialSweep(&Device, Frequency.data(), PowerSpec_dBm.data(),
&HopIndex,&FrameIndex, &MeasAuxInfo);
SWP_ResetTraceHold(&Device);
Status = Device_Close(&Device);
```

12. Phase Noise Measurement Mode

12.1 PNM_ProfileDeInit

int PNM_ProfileDeInit(void** Device, PNM_Profile_TypeDef* PNM_Profile)	
Description	
Provides the default configuration parameters for phase noise measurement, initializing the device's phase noise measurement settings.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **PNM_Profile**	Configuration structure for phase noise measurement. Refer to the definition of the PNM_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.2 PNM_Configuration

int PNM_Configuration (void** Device, const PNM_Profile_TypeDef* PNM_Profile_in, PNM_Profile_TypeDef* PNM_Profile_out, PNM_MeasInfo_TypeDef* PNM_MeasInfo)	
Description	
Configures parameters related to phase noise measurement.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **PNM_Profile_in**	Input configuration structure. Refer to the definition of the PNM_Profile_TypeDef structure for details.
[out] **PNM_Profile_out**	Output configuration structure, returning the measurement parameters actually applied by the device. Refer to the definition of the PNM_Profile_TypeDef structure for details.

[out] **PNM_MeasInfo**	Structure containing phase noise measurement information. Refer to the definition of the PNM_MeasInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_ProfileDeInit.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.3 PNM_StartMeasure

int PNM_StartMeasure(void Device)**	
Description	
Starts a phase noise measurement. Call this function to initiate the measurement before acquiring data.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_Configuration.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.4 PNM_StopMeasure

int PNM_StopMeasure(void Device)**	
Description	
Forcibly stops the current phase noise measurement.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_StartMeasure.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.5 PNM_GetPartialUpdatedFullTrace

int PNM_GetPartialUpdatedFullTrace(**void** Device,** **double* CarrierFreq,** **float* CarrierPower,** **double Freq[],** **float PhaseNoise[],** **uint32_t FrameUpdateCounts[],** **MeasAuxInfo_TypeDef* MeasAuxInfo,** **float* RefLevel**)
Description
Retrieves the phase noise measurement result trace.

Compatibility	0.55.58 and later.
Parameter description	
[in] Device	Device handle.
[out] CarrierFreq	Carrier frequency.
[out] CarrierPower	Carrier power.
[out] Freq[]	Trace frequency axis, in Hz.
[out] PhaseNoise[]	Trace power axis, in dBc/Hz.
[out] FrameUpdateCounts[]	Refresh counter (index 0 corresponds to the farthest segment, N to the nearest segment; each element represents the number of frames refreshed for that segment).
[out] MeasAuxInfo	Auxiliary measurement information structure. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
[out] RefLevel	Returns the reference level corresponding to the current measurement.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Call this function after PNM_StartMeasure. By calling the Get interface PartialUpdateCounts times, a complete measurement result can be obtained. To perform further measurements afterward, StartMeasure must be called again.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.6 PNM_AutoSearch

int PNM_AutoSearch (void** Device, double* CarrierFreq, uint8_t Found)	
Description	
Performs a panoramic scan to detect signals whose power exceeds the carrier threshold.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **CarrierFreq**	Carrier frequency.
[out] **Found**	Carrier presence flag: 0: No carrier; 1: Carrier present.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_Configuration.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.7 PNM_Preset_FrameDetRatio

int PNM_Preset_FrameDetRatio (void** Device, PNM_MeasInfo_TypeDef* MeasInfo)	
Description	
Resets the frame detection ratio for each frequency segment.	
Compatibility	0.55.58 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **MeasInfo**	After resetting the frame detection ratio, update the phase noise measurement information structure. Refer to the definition of the PNM_MeasInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_Configuration.
Example	Please refer to the relevant example of the PNM_Set_FrameDetRatio() function.

12.8 PNM_Set_FrameDetRatio

int PNM_Set_FrameDetRatio (**void** Device,** **uint32_t FrameDetRatioOfSegment[],** **PNM_MeasInfo_TypeDef* MeasInfo** **)**	
Description	
Manually configures the frame detection ratio for each frequency segment.	
Compatibility	0.55.58 and later.
Parameter description	
[in] Device	Device handle.
[in] FrameDetRatioOfSegment[]	Array for setting the frame detection ratio (array length = Segments; index 0 corresponds to the farthest segment, N to the nearest segment).
[out] MeasInfo	After adjusting the frame detection ratio, update the phase noise measurement information structure. Refer to the definition of the PNM_MeasInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after PNM_Configuration.
Example	
int Status = 0;void* Device = NULL;int DeviceNum = 0; BootProfile_TypeDef BootProfile; BootInfo_TypeDef BootInfo; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); PNM_Profile_TypeDef PNM_ProfileIn, PNM_ProfileOut; PNM_MeasInfo_TypeDef PNM_MeasInfo; PNM_ProfileDeInit(&Device, &PNM_ProfileIn); PNM_ProfileIn.CenterFreq = 1e9; PNM_ProfileIn.Threshold = -50.0; PNM_ProfileIn.RBWRatio = 0.1; PNM_ProfileIn.StartOffsetFreq = 100; PNM_ProfileIn.StopOffsetFreq = 1e6; PNM_ProfileIn.TraceAverage = 1;	

```
// Configure the device's phase noise measurement state
Status = PNM_Configuration(&Device, &PNM_ProfileIn, &PNM_ProfileOut, &PNM_MeasInfo);
// Full-band carrier search (advanced, optional): You can use the PNM_AutoSearch interface to perform a panoramic scan and detect signals exceeding the carrier decision threshold
//double PeakFreq = 1.0;
//uint8_t Found = 0;
//Status = PNM_AutoSearch(&Device, &PeakFreq, &Found);
// If an ideal carrier is found, this frequency can be configured to the device for analysis via the PNM_Configuration interface.
// Manual setting of detection ratio (advanced, optional): You can use the PNM_Set_FrameDetRatio interface to manually adjust the frame detection ratio for each frequency segment.
// PNM_MeasInfo.FrameDetRatioOfSegment[PNM_MeasInfo.Segments - 1] = 10;
// Status = PNM_Set_FrameDetRatio(&Device, PNM_MeasInfo.FrameDetRatioOfSegment, &PNM_MeasInfo);
// Reset frame detection ratio (advanced, optional): You can use the PNM_Preset_FrameDetRatio interface to restore the frame detection ratio of each segment to its default value.
// PNM_Preset_FrameDetRatio(&Device, &PNM_MeasInfo);
vector<double> Freq(PNM_MeasInfo.TracePoints);
vector<float> PhaseNoise(PNM_MeasInfo.TracePoints);
double CarrierFreq;float CarrierPower;float RefLevel;
vector<uint32_t> FrameUpdateCounts(PNM_MeasInfo.Segments);
MeasAuxInfo_TypeDef MeasAuxInfo;
while(1)
{
PNM_StartMeasure(&Device); // Start a phase noise measurement session
for (int i = 0; i < PNM_MeasInfo.PartialUpdateCounts; i++) // Retrieve the trace of a phase noise measurement result
{
Status = PNM_GetPartialUpdatedFullTrace(&Device, &CarrierFreq, &CarrierPower, Freq.data(), PhaseNoise.data(), FrameUpdateCounts.data(), &MeasAuxInfo, &RefLevel);
if (Status == APIRETVAL_NoError)
{
}
else
{
switch (Status)
{
case APIRETVAL_WARNING_CarrierLoss :
{
```

```
cout << "No carrier found" << endl; // No signal exceeding the carrier decision threshold was found
near the specified frequency points
break;
}
case APIRETVAL_WARNING_MeasUpdate :
{
i = 0;
cout << "Measurement status updated" << endl;
break;
}
default: cout << " Please refer to the programming guide to check the general error codes" << endl;
}
}
}
cout << "wait";
}
PNM_StopMeasure(&Device);
Device_Close(&Device);
```

13. Receiver/IQ Stream Main Functions

13.1 IQS_ProfileDeInit

int IQS_ProfileDeInit(void** Device, IQS_Profile_TypeDef* UserProfile_O)	
Description	
Initialize the parameters related to IQS mode. In IQS mode, parameters such as center frequency, reference level, and decimation factor are encapsulated within the IQS_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **UserProfile_O**	Pointer to the IQS configuration structure. After the function executes, this structure will be populated with system-defined default configuration values. Refer to the definition of the IQS_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the IQS_GetIQStream() function.

13.2 IQS_Configuration

int IQS_Configuration(void** Device, const IQS_Profile_TypeDef* ProfileIn, IQS_Profile_TypeDef* ProfileOut, IQS_StreamInfo_TypeDef* StreamInfo)	
Description	
Configure the spectrum analyzer for Receiver/IQ Stream (IQS) mode. In IQS mode, the local oscillator (LO) signal is fixed, and the device receives an RF signal centered at the LO frequency with a specified bandwidth. When the decimation factor is ≥2, using USB 3.0 and a high-speed hard drive for data transfer enables continuous time-domain recording.	
Compatibility	0.55.77 and later.
Parameter description	

[in] **Device**	Device handle.
[in] **IQS_ProfileIn**	Pointer to the IQS configuration structure. Refer to the definition of the IQS_Profile_TypeDef structure for details.
[out] **IQS_ProfileOut**	Pointer to the IQS configuration structure. Not all data in ProfileIn may be strictly applied; the device will automatically adjust certain parameters based on hardware capabilities or internal constraints. The actual applied settings are reflected in the output configuration set. Refer to the definition of the IQS_Profile_TypeDef structure for details.
[out] **StreamInfo**	Information related to the IQ time-domain data stream in IQS mode. Refer to the definition of the IQS_StreamInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_ProfileDeInit.
Example	Please refer to the relevant example of the IQS_GetIQStream() function.

13.3 IQS_BusTriggerStart

int IQS_BusTriggerStart(void Device)**	
Description	
Initiate a bus trigger.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before IQS_GetIQStream.
Example	Please refer to the relevant example of the IQS_GetIQStream() function.

13.4 IQS_BusTriggerStop

int IQS_BusTriggerStop(void Device)**	
Description	
Terminate the current bus trigger. When TriggerMode = FixedPoints, the bus trigger will automatically stop after being started by the IQS_BusTriggerStart function and reaching the specified trigger length, without needing to call this function.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before IQS_GetIQStream.
Example	Please refer to the relevant example of the IQS_GetIQStream() function.

13.5 IQS_GetIQStream

int IQS_GetIQStream(void** Device, void** AlternIQStream, float* ScaleToV, IQS_TriggerInfo_TypeDef* TriggerInfo, MeasAuxInfo_TypeDef* MeasAuxInfo)	
Description	
Call this function to obtain time-domain data, measurement information, and trigger information in IQS mode.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **AlternIQStream**	Address of the time-domain data (interleaved IQ format). Each data packet is fixed at 64,968 bytes. When the IQ data type is int8_t, each of the I and Q channels contains 32,484 points, with each point occupying 1 byte; When the IQ data type is int16_t, each of the I and Q channels contains 16,242 points, with each point occupying 2 bytes; When the IQ data type is int32_t, each of the I and Q channels contains 8,121 points, with each point occupying 4 bytes. IQ data is stored in an interleaved manner: I Q I Q ...
[out] **ScaleToV**	Coefficient for converting the raw time-domain data acquired by the ADC into absolute voltage (V).
[out] **TriggerInfo**	Trigger-related information for the IQ data stream. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
[out] **MeasAuxInfo**	Returns auxiliary information for the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_Configuration.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB;	

```
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
Status = IQS_BusTriggerStart(&Device);
void* AlternIQStream = NULL;
float ScaleToV = 0;
IQS_TriggerInfo_TypeDef TriggerInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = IQS_GetIQStream(&Device, &AlternIQStream, &ScaleToV, &TriggerInfo, &MeasAuxInfo);
Status = IQS_BusTriggerStop(&Device);
Status = Device_Close(&Device);
```

14. Receiver/IQ Stream Other Functions

14.1 IQS_MultiDevice_WaitExternalSync

int IQS_MultiDevice_WaitExternalSync(void** Device, const IQS_Profile_TypeDef* ProfileIn)	
Description	
Call this function to wait for a multi-device synchronous trigger signal, allowing multiple devices to start or acquire data simultaneously.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **ProfileIn**	Pointer to the IQS configuration structure. Refer to the definition of the IQS_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This must be called after IQS_Configuration, with TriggerSource set to MultiDevSyncByExt or MultiDevSyncByGNSS1PPS.
Example	Please refer to the relevant example of the IQS_MultiDevice_Run() function.

14.2 IQS_MultiDevice_Run

int IQS_MultiDevice_Run(void** Device)	
Description	
Call this function to enable multi-device synchronous operation, starting multiple devices for synchronized data acquisition or operation.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_MultiDevice_WaitExternalSync.
Example	

int Status = -1, Status0 = -1; int DeviceNum0 = 0; int DeviceNum1 = 0;
void* Device0 = NULL; void* Device1 = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;	

```
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device0, DeviceNum0, &BootProfile, &BootInfo);
Status0 = Device_Open(&Device1, DeviceNum1, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn0, ProfileIn1;
IQS_Profile_TypeDef ProfileOut0, ProfileOut1;
IQS_StreamInfo_TypeDef StreamInfo0, StreamInfo1;
Status = IQS_ProfileDeInit(&Device0, &ProfileIn0);
Status = IQS_ProfileDeInit(&Device1, &ProfileIn1);
ProfileIn0.TriggerSource = MultiDevSyncByExt;
ProfileIn1.TriggerSource = MultiDevSyncByExt;
Status = IQS_Configuration(&Device0, &ProfileIn0, &ProfileOut0, &StreamInfo0);
Status0 = IQS_Configuration(&Device1, &ProfileIn1, &ProfileOut1, &StreamInfo1);
Status0 = IQS_MultiDevice_WaitExternalSync(&Device0, &ProfileOut0);
Status0 = IQS_MultiDevice_Run(&Device0);
Status = IQS_MultiDevice_Run(&Device1);
Status = Device_Close(&Device0);
Status0 = Device_Close(&Device1);
```

14.3 IQS_SyncTimer

int IQS_SyncTimer(void Device)**	
Description	
Call this function to initiate timer-to-external-trigger synchronization.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_Configuration, with TriggerSource set to Timer.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
ProfileIn.TriggerSource = Timer;
```

ProfileIn. TriggerTimerSync = SyncToExt_RisingEdge; Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo); Status = IQS_SyncTimer (&Device); Status = Device_Close(&Device);

14.4 IQS_GetIQStream_PM1

int IQS_GetIQStream_PM1(**void** Device,** **IQStream_TypeDef* IQStream** **)**	
Description	
Retrieve time-domain data in IQS mode. The data format can be int8_t, int16_t, or int32_t, which the user can select according to actual requirements.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] IQStream	IQ data stream, including IQ data and related configuration information. Refer to the definition of the IQStream_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_Configuration.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); IQS_Profile_TypeDef ProfileIn, ProfileOut; IQS_StreamInfo_TypeDef StreamInfo; Status = IQS_ProfileDeInit(&Device, &ProfileIn); Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo); IQStream_TypeDef IQStream; Status = IQS_BusTriggerStart(&Device); Status = IQS_GetIQStream_PM1(&Device, &IQStream); Status = IQS_BusTriggerStop(&Device); Status = Device_Close(&Device);	

14.5 IQS_GetIQStream_PM2

int IQS_GetIQStream_PM2(**void** Device,** **IQStream_TypeDef* IQStream,** **MeasAuxInfo_TypeDef* MeasAuxInfo** **)**
Description
Retrieve time-domain data and measurement auxiliary information in IQS mode. The time-domain data format can be int8_t, int16_t, or int32_t, and the user can choose the format as needed.

Compatibility	0.55.0 and later.
Parameter description	
[in] Device	Device handle.
[out] IQStream	IQ data stream, including the IQ data and associated configuration information. Refer to the definition of the IQStream_TypeDef structure for details.
[out] MeasAuxInfo	Returns auxiliary information for the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_Configuration.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
IQStream_TypeDef IQStream;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = IQS_BusTriggerStart(&Device);
Status = IQS_GetIQStream_PM2(&Device, &IQStream, &MeasAuxInfo);
Status = IQS_BusTriggerStop(&Device);
Status = Device_Close(&Device);
```

14.6 IQS_GetIQStream_Data

int IQS_GetIQStream_Data(void** Device, int16_t IQ_data[])	
Description	
Call this function to obtain IQ time-domain data in int16_t format.	
Compatibility	0.55.0 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **IQ_data[]**	Array for receiving single-channel 16-bit IQ data.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after IQS_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
Status = IQS_BusTriggerStart(&Device);
vector<int16_t> IQ_Data(StreamInfo.StreamSamples);
Status = IQS_GetIQStream_Data(&Device, IQ_Data.data());
Status = Device_Close(&Device);
```

15. Power Detection Mode

DET is a detection analysis mode that performs power detection on signals within a specified bandwidth, helping users observe signal amplitude levels.

15.1 DET_ProfileDeInit

int DET_ProfileDeInit(**void** Device,** **DET_Profile_TypeDef* UserProfile_O**)	
Description	
Initialize parameters related to DET mode. In DET mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the DET_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] UserProfile_O	Pointer to the DET configuration structure. Refer to the definition of the DET_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the DET_GetPowerStream() function.

15.2 DET_Configuration

int DET_Configuration(**void** Device,** **const DET_Profile_TypeDef* ProfileIn,** **DET_Profile_TypeDef* ProfileOut,** **DET_StreamInfo_TypeDef* StreamInfo**)	
Description	
Configure parameters related to DET mode. In DET mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the DET_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.

[in] **DET_ProfileIn**	Pointer to the DET configuration structure. Refer to the definition of the DET_Profile_TypeDef structure for details.
[out] **DET_ProfileOut**	Pointer to the DET configuration structure. Refer to the definition of the DET_Profile_TypeDef structure for details.
[out] **StreamInfo**	Information related to DET data in DET mode. Refer to the definition of the DET_StreamInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DET_ProfileDeInit.
Example	Please refer to the relevant example of the DET_GetPowerStream() function.

15.3 DET_BusTriggerStart

int DET_BusTriggerStart(void Device)**	
Description	
Initiate a bus trigger.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before DET_GetPowerStream.
Example	Please refer to the relevant example of the DET_GetPowerStream() function.

15.4 DET_BusTriggerStop

int DET_BusTriggerStop(void Device)**	
Description	
Terminate the current bus trigger. When TriggerMode = FixedPoints, the bus trigger will automatically stop after being started by the DET_BusTriggerStart function and reaching the specified trigger length, without needing to call this function.	
Compatibility	0.55.0 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before DET_GetPowerStream.
Example	Please refer to the relevant example of the DET_GetPowerStream() function.

15.5 DET_GetPowerStream

int DET_GetPowerStream(**void** Device,** **float NormalizedPowerStream[],** **float* ScaleToV,** **DET_TriggerInfo_TypeDef* TriggerInfo,** **MeasAuxInfo_TypeDef* MeasAuxInfo** **)**
Description
Retrieve detection data in DET mode, along with the conversion factor from integer values to absolute amplitude (in volts) and trigger-related information. NormalizedPowerStream is calculated as $\sqrt{(I^2 + Q^2)}$.

Compatibility	0.55.0 and later.
Parameter description	
[in] Device	Device handle.
[out]NormalizedPowerStream[]	Returns the instantaneous amplitude of each data point, calculated as $\sqrt{(I^2 + Q^2)}$.
[out] ScaleToV	Coefficient for converting unitless instantaneous amplitude (NormalizedPowerStream) to absolute voltage (V).
[out] TriggerInfo	Trigger-related information for DET data. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
[out] MeasAuxInfo	Returns auxiliary information for the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DET_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
DET_Profile_TypeDef ProfileIn, ProfileOut;
DET_StreamInfo_TypeDef StreamInfo;
```

```
Status = DET_ProfileDeInit(&Device, &ProfileIn);
Status = DET_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
vector<float> NormalizedPowerStream(StreamInfo.PacketSamples);
float ScaleToV = 0;
DET_TriggerInfo_TypeDef TriggerInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = DET_BusTriggerStart(&Device);
Status = DET_GetPowerStream(&Device, NormalizedPowerStream.data(), &ScaleToV, &TriggerInfo,
&MeasAuxInfo);
Status = DET_BusTriggerStop(&Device);
Status = Device_Close(&Device);
```

15.6 DET_SyncTimer

int DET_SyncTimer(void Device)**	
Description	
Call this function to initiate timer-to-external-trigger synchronization.	
Compatibility	0.55.0 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DET_Configuration, with TriggerSource set to Timer.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
DET_Profile_TypeDef ProfileIn, ProfileOut;
DET_StreamInfo_TypeDef StreamInfo;
Status = DET_ProfileDeInit(&Device, &ProfileIn);
ProfileIn.TriggerSource = Timer;
ProfileIn. TriggerTimerSync = SyncToExt_RisingEdge;
Status = DET_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
Status = DET_SyncTimer (&Device);
Status = Device_Close(&Device);
```

16. Zero Span Mode

Zero Span mode is used for zero-span analysis of a signal, providing real-time power measurements at a specific frequency to help users accurately capture instantaneous signal variations.

16.1 ZSP_ProfileDeInit

int ZSP_ProfileDeInit(**void** Device,** **ZSP_Profile_TypeDef* UserProfile_O**)	
Description	
Initialize parameters related to Zero Span mode. In Zero Span mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the ZSP_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] UserProfile_O	Pointer to the Zero Span configuration structure. Refer to the definition of the ZSP_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the ZSP_Configuration() function.

16.2 ZSP_Configuration

int ZSP_Configuration(**void** Device,** **const ZSP_Profile_TypeDef* ProfileIn,** **ZSP_Profile_TypeDef* ProfileOut,** **DET_StreamInfo_TypeDef* StreamInfo**)	
Description	
Configure parameters related to ZeroSpan mode. In ZeroSpan mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the ZSP_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	

[in] Device	Device handle.
[in] ProfileIn	Pointer to the Zero Span configuration structure. Refer to the definition of the ZSP_Profile_TypeDef structure for details.
[out] ProfileOut	Pointer to the Zero Span configuration structure. Refer to the definition of the ZSP_Profile_TypeDef structure for details.
[out] StreamInfo	Information related to ZSP data in Zero Span mode. Refer to the definition of the DET_StreamInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after ZSP_ProfileDeInit.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
ZSP_Profile_TypeDef ProfileIn, ProfileOut;
DET_StreamInfo_TypeDef StreamInfo;
Status = ZSP_ProfileDeInit(&Device, &ProfileIn);
ProfileIn.CenterFreq_Hz = 1e9;
ProfileIn.DecimateFactor = 2;
Status = ZSP_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
Status = Device_Close(&Device);
```

17. Real-Time Spectrum Analysis Mode

RTA is the Real-Time Spectrum Analysis mode, which helps users observe frequency-hopping or short transient burst signals.

17.1 RTA_ProfileDeInit

int RTA_ProfileDeInit(**void** Device,** **RTA_Profile_TypeDef* UserProfile_O**)	
Description	
Initialize parameters related to RTA mode. In RTA mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the RTA_Profile_TypeDef structure	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.
[out] UserProfile_O	Pointer to the RTA configuration structure. Refer to the definition of the RTA_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the RTA_GetRealTimeSpectrum() function.

17.2 RTA_Configuration

int RTA_Configuration(**void** Device,** **const RTA_Profile_TypeDef* ProfileIn,** **RTA_Profile_TypeDef* ProfileOut,** **RTA_FrameInfo_TypeDef* FrameInfo**)	
Description	
Configure parameters related to RTA mode. In RTA mode, parameters such as center frequency, reference level, and decimation factor are encapsulated in the RTA_Profile_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] Device	Device handle.

[in] **RTA_ProfileIn**	Pointer to the RTA configuration structure. Refer to the definition of the RTA_Profile_TypeDef structure for details.
[out] **RTA_ProfileOut**	Pointer to the RTA configuration structure. Refer to the definition of the RTA_Profile_TypeDef structure for details.
[out] **FrameInfo**	Packet information structure returned after configuration in RTA mode. Refer to the definition of the RTA_FrameInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the RTA_GetRealTimeSpectrum() function.

17.3 RTA_BusTriggerStart

int RTA_BusTriggerStart(void Device)**	
Description	
Initiate a bus trigger.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before RTA_GetRealTimeSpectrum.
Example	Please refer to the relevant example of the RTA_GetRealTimeSpectrum() function.

17.4 RTA_BusTriggerStop

int RTA_BusTriggerStop(void Device)**	
Description	
Terminate the current bus trigger. When TriggerMode = FixedPoints, the bus trigger will automatically stop after being started by the RTA_BusTriggerStart function and reaching the specified trigger length, without needing to call this function.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after RTA_GetRealTimeSpectrum.
Example	Please refer to the relevant example of the RTA_GetRealTimeSpectrum() function.

17.5 RTA_SetDataFormat

int RTA_SetDataFormat(void** Device, DataFormat_TypeDef* DataFormat)	
Description	
Set the data type for real-time spectrum.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **DataFormat**	Set the data type for real-time spectrum. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before RTA_Configuration.
Example	Please refer to the relevant example of the RTA_GetIQStream() function.

17.6 RTA_SetLookBackCmd

int RTA_SetLookBackCmd(void** Device, LookBack_TypeDef* LookBackCmd)	
Description	
Set the LookBack state of the real-time spectrum. When LookBack is enabled, the device caches the IQ data corresponding to the spectrum, allowing the user to retrieve it when needed.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[in] **LookBackCmd**	Set the LooBack state of the real-time spectrum. Refer to the definition of the LookBack_TypeDef enumeration structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before RTA_Configuration.
Example	Please refer to the relevant example of the RTA_GetIQStream() function.

17.7 RTA_TriggerStart

int RTA_TriggerStart(void** Device)	
Description	
When the trigger mode is bus trigger, calling this function will immediately initiate the trigger. When the trigger mode is non-bus trigger, this function enables the trigger and waits for the trigger event to occur before starting device acquisition.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before RTA_GetRealTimeSpectrum.
Example	Please refer to the relevant example of the RTA_GetIQStream() function.

17.8 RTA_GetIQStream

```
int RTA_GetIQStream(
    void** Device,
    void* AlternIQStream,
    float* ScaleToV,
    IQS_TriggerInfo_TypeDef* TriggerInfo,
    MeasAuxInfo_TypeDef* MeasAuxInfo
)
```

Description	
Retrieve the real-time IQStream and trigger-related information in RTA mode.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **AlternIQStream**	Time-domain data (IQ data is stored in an interleaved format as IQIQ...). Each data packet can contain up to 8192 IQ points. The total number of bytes in a data packet is determined by the InPacketTriggeredDataSize parameter in the IQS_TriggerInfo_TypeDef structure. When the data type is int16_t, the total number of data points equals InPacketTriggeredDataSize / 2. When the data type is int8_t, the total number of data points equals InPacketTriggeredDataSize.

	When the data type is int32_t, the total number of data points equals InPacketTriggeredDataSize / 4.
[out] **ScaleToV**	Coefficient for converting the raw time-domain data acquired by the ADC into the absolute voltage value (V).
[out] **TriggerInfo**	Trigger-related information of the IQ data stream. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
[out] **MeasAuxInfo**	Return auxiliary information related to the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before RTA_GetRealTimeSpectrum.

Example

```
int Status = -1; int DeviceNum = 0; void *Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
RTA_Profile_TypeDef RTA, RTA_Feedback;
RTA_FrameInfo_TypeDef FrameInfo;
RTA_PlotInfo_TypeDef RTA_PlotInfo;
RTA_TriggerInfo_TypeDef TriggerInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
float ScaleToV;
int32_t pointsize = 1;
vector<uint8_t> AlternIQStream(131072);
DataFormat_TypeDef DataFormat = Real16bit;
LookBack_TypeDef LookBackCmd = LookBack_On;
RTA_ProfileDeInit(&Device, &RTA);
RTA.DecimateFactor = 32;
RTA.RBWMode = RBW_Manual;
RTA.RBW_Hz = 1e6;
RTA.SweepTime = 0.0001;
RTA.TriggerAcqTime = 0.1;
if (DataFormat == Real8bit)
    pointsize = 1;
else if (DataFormat == Real16bit)
    pointsize = 2;
```

```
Status = RTA_SetDataFormat(&Device, &DataFormat);
Status = RTA_SetLookBackCmd(&Device, &LookBackCmd);
Status = RTA_Configuration(&Device, &RTA, &RTA_Feedback, &FrameInfo);
vector<uint8_t> SpectrumTrace(FrameInfo.PacketValidPoints *pointsize);
vector<uint16_t> SpectrumBitmap(FrameInfo.FrameHeight *FrameInfo.FrameWidth);
while (1)
{
    RTA_TriggerStart(&Device);
    for (int i = 0; i < FrameInfo.PacketCount; i++) {
        Status = RTA_GetRealTimeSpectrum(&Device, SpectrumTrace.data(),
SpectrumBitmap.data(), &RTA_PlotInfo, &TriggerInfo, &MeasAuxInfo);
    }
    for (int i = 0; i < FrameInfo.PacketCount; i++) {
        Status = RTA_GetIQStream(&Device, (void *)AlternIQStream.data(), &ScaleToV,
&TriggerInfo, &MeasAuxInfo);
        if (Status == APIRETVAL_LastPacket) {
            break;
        }
    }
}
Device_Close(&Device);
```

17.9 RTA_GetRealTimeSpectrum_Raw

```
int RTA_GetRealTimeSpectrum_Raw(
    void** Device,
    uint8_t SpectrumStream[],
    RTA_PlotInfo_TypeDef* PlotInfo,
    RTA_TriggerInfo_TypeDef* TriggerInfo,
    MeasAuxInfo_TypeDef* MeasAuxInfo
)
```

Description

Retrieve real-time spectrum data in RTA mode (without probability density plot) along with trigger-related information.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **SpectrumStream[]**	Returns a spectrum stream composed of consecutive spectrum frames, represented as relative power, with LSB = 0.75 dB. The size of this array

	equals RTA_FrameInfo.PacketValidPoints obtained from the RTA_Configuration function.
[out] RTA_PlotInfo	Structure containing plotting information returned after RTA acquisition. Refer to the definition of the RTA_PlotInfo_TypeDef structure for details.
[out] TriggerInfo	Trigger-related information for RTA data. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
[out] MeasAuxInfo	Returns auxiliary information for the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after RTA_Configuration.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
RTA_Profile_TypeDef ProfileIn, ProfileOut;
RTA_FrameInfo_TypeDef FrameInfo;
Status = RTA_ProfileDeInit(&Device, &ProfileIn);
Status = RTA_Configuration(&Device, &ProfileIn, &ProfileOut, &FrameInfo);
vector<uint8_t> SpectrumTrace(FrameInfo.PacketValidPoints);
RTA_TriggerInfo_TypeDef TriggerInfo;
RTA_PlotInfo_TypeDef PlotInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = RTA_BusTriggerStart(&Device);
Status = RTA_GetRealTimeSpectrum_Raw(&Device, SpectrumTrace.data(), &PlotInfo, &TriggerInfo,
&MeasAuxInfo);
Status = RTA_BusTriggerStop(&Device);
Status = Device_Close(&Device);
```

17.10 RTA_GetRealTimeSpectrum

```
int RTA_GetRealTimeSpectrum(
    void** Device,
    uint8_t SpectrumStream[],
    uint16_t SpectrumBitmap[],
    RTA_PlotInfo_TypeDef* PlotInfo,
    RTA_TriggerInfo_TypeDef* TriggerInfo,
    MeasAuxInfo_TypeDef* MeasAuxInfo
)
```

Description	
Retrieve real-time spectrum data and trigger-related information in real-time spectrum mode.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **SpectrumStream[]**	Returns a spectrum stream composed of consecutive spectrum frames, represented as relative power with LSB = 0.75 dB. The size of this array equals RTA_FrameInfo.PacketValidPoints obtained from the RTA_Configuration function.
[out] **SpectrumBitmap[]**	Returns the probability density map bitmap. Array length = FrameInfo.FrameHeight × FrameInfo.FrameWidth。
[out] **RTA_PlotInfo**	Structure containing plotting information returned after RTA acquisition Refer to the definition of the RTA_PlotInfo_TypeDef structure for details.
[out] **TriggerInfo**	Trigger-related information for RTA data. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
[out] **MeasAuxInfo**	Returns auxiliary information for the measurement data. Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after RTA_Configuration.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
RTA_Profile_TypeDef ProfileIn, ProfileOut;
```

```
RTA_FrameInfo_TypeDef FrameInfo;
Status = RTA_ProfileDeInit(&Device, &ProfileIn);
Status = RTA_Configuration(&Device, &ProfileIn, &ProfileOut, &FrameInfo);
vector<uint8_t> SpectrumTrace(FrameInfo.PacketValidPoints);
vector<uint16_t> SpectrumBitmap(FrameInfo.FrameHeight * FrameInfo.FrameWidth);
RTA_TriggerInfo_TypeDef TriggerInfo;
RTA_PlotInfo_TypeDef PlotInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = RTA_BusTriggerStart(&Device);
Status = RTA_GetRealTimeSpectrum(&Device, SpectrumTrace.data(), SpectrumBitmap.data(),
&PlotInfo, &TriggerInfo, &MeasAuxInfo);
Status = RTA_BusTriggerStop(&Device);
Status = Device_Close(&Device);
```

17.11 RTA_SyncTimer

int RTA_SyncTimer(void Device)**	
Description	
Call this function to initiate a single timer-to-external-trigger synchronization.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This must be called after RTA_Configuration, with TriggerSource set to Timer.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
RTA_Profile_TypeDef ProfileIn, ProfileOut;
RTA_FrameInfo_TypeDef FrameInfo;
Status = RTA_ProfileDeInit(&Device, &ProfileIn);
ProfileIn.TriggerSource = Timer;
ProfileIn. TriggerTimerSync = SyncToExt_RisingEdge;
Status = RTA_Configuration(&Device, &ProfileIn, &ProfileOut, &FrameInfo);
Status = RTA_SyncTimer (&Device);
Status = Device_Close(&Device);
```

18. MSCAN Mode

18.1 MSCAN_ProfileDeinit

int MSCAN_ProfileDeinit(void** Device, MSCAN_Profile_TypeDef* MSCAN_Profiles, int32_t* Elements)	
Description	
Initializes parameters related to storing the scan mode configuration. In MSCAN mode, parameters such as center frequency, reference level, and dwell time are all encapsulated in the MSCAN_Profile_TypeDef structure.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **MSCAN_Profiles**	Pointer to the MSCAN configuration structure. Refer to the definition of the MSCAN_Profile_TypeDef structure for details.
[out] **Elements**	Number of MSCAN configuration files.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the MSCAN_GetData() function.

18.2 MSCAN_Configuration

int MSCAN_Configuration(void** Device, MSCAN_Profile_TypeDef* ProfilesIn, MSCAN_Profile_TypeDef* ProfilesOut, MSCAN_Info_Typedef* MSCAN_Info, int32_t* Elements, int64_t* Repeatitions, PreamplifierState_TypeDef* Preamplifier)	
Description	
Configure the MSCAN sweep list, sweep count, and related settings.	
Compatibility	0.55.79 and later.

Parameter description	
[in] **Device**	Device handle.
[in] **ProfilesIn**	Pointer to the MSCAN configuration structure. Refer to the definition of the MSCAN_Profile_TypeDef structure for details.
[out] **ProfilesOut**	Pointer to the MSCAN configuration structure. Refer to the definition of the MSCAN_Profile_TypeDef structure for details.
[out] **MSCAN_Info**	Return the MSCAN spectrum frame count, spectrum point count, and IQ data point count. Refer to the definition of the MSCAN_Info_Typedef structure for details.
[out] **Elements**	Number of MSCAN configuration files. The maximum value is 128; any value exceeding 128 will be overwritten as 128.
[out] **Repeatitions**	Pointer to the repeat count of each MSCAN configuration file in the scan list.
[out] **Preamplifier**	Pointer to the preamplifier structure. Refer to the definition of the PreamplifierState_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after MSCAN_ProfileDeinit.
Example	Please refer to the relevant example of the MSCAN_GetData() function.

18.3 MSCAN_Start

int MSCAN_Start(void Device)**	
Description	
Strat the list sweep according to the user-specified number of the repetitions. The sweep automatically stops upon completion.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after MSCAN_Configuration.
Example	Please refer to the relevant example of the MSCAN_GetData() function.

18.4 MSCAN_Stop

int MSCAN_Stop(void** Device)	
Description	
Stop list sweep.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after MSCAN_GetData.
Example	Please refer to the relevant example of the MSCAN_GetData() function.

18.5 MSCAN_GetData

int MSCAN_GetData(void** Device, MSCAN_Data_Typedef* MSCAN_Data)	
Description	
Retrieve the sweep results.	
Compatibility	0.55.79 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **MSCAN_ Data**	Pointer to the MSCAN scan result structure. After calling MSCAN_GetData, this structure contains the measurement data corresponding to each scan configuration file. Refer to the definition of the MSCAN_Data_Typedef structure for details.
Return value.	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after MSCAN_Start.
Example	
int Status = 0; int DeviceNum = 0; void *Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);	

```
int32_t Elements = 95;
int64_t Repeatitions = 100;
std::vector<MSCAN_Profile_TypeDef> MSCAN_ProfileIn(Elements), MSCAN_ProfileOut(Elements);
std::vector<MSCAN_Info_Typedef> MSCAN_Info(Elements);
Status = MSCAN_ProfileDeinit(&Device, MSCAN_ProfileIn.data(), &Elements);

for (uint32_t i = 0; i < Elements; i++)
{
	MSCAN_ProfileIn[i].CenterFreq_Hz = 50e6 + i * 100e6;
	MSCAN_ProfileIn[i].DwellTime = 16.384e-6;
	MSCAN_ProfileIn[i].DetectCount = 1;
	MSCAN_ProfileIn[i].IQPlayBack = IQPlayBack_Off;
	MSCAN_ProfileIn[i].FFTSize = 2048;
}
PreamplifierState_TypeDef Preamplifier = AutoOn;
Status = MSCAN_Configuration(&Device, MSCAN_ProfileIn.data(), MSCAN_ProfileOut.data(),
MSCAN_Info.data(), &Elements, &Repeatitions, &Preamplifier);

// Allocate buffer sized for the largest element.
uint32_t SpectrumFrames_max = 0;
uint32_t SpectrumPoints_max = 0;
uint32_t IQStreamPoints_max = 0;
for (uint32_t i = 0; i < Elements; i++) {
	if (SpectrumFrames_max < MSCAN_Info[i].SpectrumFrames) {
		SpectrumFrames_max = MSCAN_Info[i].SpectrumFrames;
	}
	if (SpectrumPoints_max < MSCAN_Info[i].SpectrumPoints) {
		SpectrumPoints_max = MSCAN_Info[i].SpectrumPoints;
	}
	if (IQStreamPoints_max < MSCAN_Info[i].IQStreamPoints) {
		IQStreamPoints_max = MSCAN_Info[i].IQStreamPoints;
	}
}
std::vector<uint8_t> SpectrumStream(SpectrumFrames_max *SpectrumPoints_max);
std::vector<double> Frequency(SpectrumPoints_max);
std::vector<int16_t> IQStream(IQStreamPoints_max * 2);
MSCAN_Data_Typedef MSCAN_Data;
MSCAN_Data.SpectrumStream = SpectrumStream.data();
MSCAN_Data.IQStream = IQStream.data();
```

```
while (1) {
        MSCAN_Start(&Device);
        auto start = std::chrono::high_resolution_clock::now();
        for (uint32_t i = 0; i < Repeatitions; i++) {
                for (uint32_t t = 0; t < Elements;) {
                        Status = MSCAN_GetData(&Device, &MSCAN_Data);
                        if (!Status) {
                                t = MSCAN_Data.ElementIndex + 1;
                        }
                        if (MSCAN_Data.ElementIndex == (Elements - 1) && MSCAN_Data.RepeatIndex
  == (Repeatitions - 1)) {
                                break;
                        }
                }
                if (MSCAN_Data.ElementIndex == (Elements - 1) && MSCAN_Data.RepeatIndex ==
(Repeatitions - 1)) {
                        break;
                }
        }
        auto stop = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::duration<double, std::ratio<1,
1000>>>(stop - start);
}
Device_Close(&Device);
```

19. Digital Demodulation (option)

Consists of the modulated signal spectrum, demodulated constellation diagram, eye diagram, and demodulation parameters. It provides an in-depth analysis of signal modulation quality, offering multiple error metrics to effectively evaluate the integrity and reliability of the signal during transmission.

19.1 Demod_Check

int Demod_Check()	
Description	
Verify whether the digital demodulation library is present	
Compatibility	0.55.55 and later.
Return value	0: Demodulation library is present; -1: Demodulation library is not present.
Calling constraints	None.
Example	
int Status = -1; Status = Demod_Check();	

19.2 Demod_Open

int Demod_Open(void Device)**	
Description	
Enable the demodulation function, checking for the presence of a license and allocating the required memory.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.3 Demod_Close

int Demod_Close(void Device)**	
Description	
Disable the demodulation function.	
Compatibility	0.55.55 and later.
Parameter description	

[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Demod_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.4 Demod_Reset

int Demod_Reset(void Device)**	
Description	
Reset the demodulation function. When IQ data is non-continuous, Demod_Reset must be called before each Demod_Execute call. If the IQ data is continuous, Demod_Execute can be called repeatedly without resetting.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **Device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Demod_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.5 Demod_GetVersion

int Demod_Reset(**void** Device, **char version[]** **)**	
Description	
Retrieve the demodulation API version.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **version[]**	Returns the demodulation API version.
Return value	Returns the character length of version.
Calling constraints	Must be called after Demod_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.6 Demod_DeInit

void Demod_DeInit(Demod_Profile_TypeDef* DemodProfile)	
Description	
Initialize the demodulation configuration structure, assigning default values to each parameter within the structure.	
Compatibility	0.55.55 and later.
Parameter description	
[in] DemodProfile	Demodulation configuration structure. Refer to the definition of the Demod_Profile_TypeDef structure for details.
Return value	None.
Calling constraints	Must be called after Demod_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.7 Demod_Configuration

int Demod_Configuration(**void** ** **Device,** **const Demod_Profile_TypeDef* DemodProfileIn,** **Demod_Profile_TypeDef* DemodProfileOut** **)**	
Description	
Configure demodulation parameters.	
Compatibility	0.55.55 and later.
Parameter description	
[in] Device	Device handle.
[in] DemodProfileIn	Input demodulation configuration structure. Refer to the definition of the Demod_Profile_TypeDef structure for details.
[out] DemodProfileOut	Output demodulation configuration structure. If the input configuration is invalid, the structure will be updated with valid parameters. Refer to the definition of the Demod_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Demod_Open.
Example	Please refer to the relevant example of the Demod_Execute() function.

19.8 Demod_Execute

int Demod_Execute(**void** Device,** **const IQStream_TypeDef* IQStream,** **DemodInfo_TypeDef* DemodInfo** **)**	
Description	
Set demodulation parameters.	
Compatibility	0.55.55 and later.
Parameter description	
[in] Device	Device handle.
[in] IQStream	IQ data stream, including the IQ data and associated configuration information. Refer to the definition of the IQStream_TypeDef structure for details.
[out] DemodInfo	Demodulation information structure, including eye diagram, constellation diagram, EVM, etc. Note: All pointer variables in the structure point to internal memory managed by the function; users do not need to allocate memory externally. Refer to the definition of the DemodInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Demod_Open.
Example	

int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef IQS_ProfileIn, IQS_ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
IQS_TriggerInfo_TypeDef TriggerInfo;
IQS_ProfileDeInit(&Device, &IQS_ProfileIn);
IQS_ProfileIn.CenterFreq_Hz = 1e9;
IQS_ProfileIn.RefLevel_dBm = 0;
//Note: The demodulation function currently only accepts IQ data of type int16.
IQS_ProfileIn.DataFormat = Complex16bit;
IQS_ProfileIn.TriggerMode = FixedPoints;
IQS_ProfileIn.TriggerSource = Bus;	

```
IQS_ProfileIn.DecimateFactor = 4;
IQS_ProfileIn.TriggerLength = 32484;
Status = IQS_Configuration(&Device, &IQS_ProfileIn, &IQS_ProfileOut, &StreamInfo);
IQStream_TypeDef IQStream;
vector<int16_t> IQ(StreamInfo.StreamSamples * 2);
vector<int16_t> I(StreamInfo.StreamSamples);
vector<int16_t> Q(StreamInfo.StreamSamples);
Status = Demod_Open(&Device);
vector<char> version(50);
Demod_GetVersion(&Device, version.data());
Demod_Profile_TypeDef DemodProfileIn, DemodProfileOut;
DemodInfo_TypeDef DemodInfo;
Demod_DeInit(&DemodProfileIn);
DemodProfileIn.SamplePoints = StreamInfo.StreamSamples;
DemodProfileIn.SampleRate = StreamInfo.IQSampleRate;
DemodProfileIn.ModType = QAM16;
DemodProfileIn.SymbolRate = 1e6;
Status = Demod_Configuration(&Device, &DemodProfileIn, &DemodProfileOut);
IQS_ProfileIn.TriggerLength = DemodProfileOut.SamplePoints;
Status = IQS_Configuration(&Device, &IQS_ProfileIn, &IQS_ProfileOut, &StreamInfo);
DemodProfileIn.SamplePoints = StreamInfo.StreamSamples;
Status = Demod_Configuration(&Device, &DemodProfileIn, &DemodProfileOut);
while (1) {
uint32_t Points = StreamInfo.PacketSamples;
Status = IQS_BusTriggerStart(&Device);
for (uint32_t i = 0; i < StreamInfo.PacketCount; i++) {
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
if (i == StreamInfo.PacketCount - 1 && StreamInfo.StreamSamples % StreamInfo.PacketSamples != 0) {
Points = StreamInfo.StreamSamples % StreamInfo.PacketSamples;
}
memcpy(IQ.data() + i * StreamInfo.PacketSamples * 2, IQStream.AlternIQStream, Points * 2 *
sizeof(IQ[0]));
}
IQStream.AlternIQStream = IQ.data();
Demod_Reset(&Device);
Status = Demod_Execute(&Device, &IQStream, &DemodInfo);
}
Status = Demod_Close(&Device);
Status = Device_Close(&Device);
```

19.9 Demod_GenSymbolMap

void Demod_GenSymbolMap(Demod_ModType_TypeDef **ModType,** Demod_SymbolMap_TypeDef **SymbolMap[1024],** uint32_t* **MapNum**)	
Description	
Generate a symbol mapping table based on the input modulation type, and calculate the number of symbols in the table.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **ModType**	Modulation type. Including 2FSK, 4FSK, GMSK, BPSK, QPSK, 8PSK, 16QAM, 2ASK, 64QAM, AM, FM, PM, CW, Lower Sideband, Upper Sideband, 128QAM, 256QAM, 32QAM.
[out] **SymbolMap[1024]**	Represents a single symbol in the symbol mapping table. This symbol's mapping defines the relationship between the symbol and its coordinates used in the modulation and demodulation process. Refer to the definition of the Demod_SymbolMap_TypeDef structure for details.
[out] **MapNum**	The number of available symbols in the symbol mapping table.
Return value	None.
Calling constraints	None.
Example	

int Status = -1;
Demod_ModType_TypeDef ModType = QPSK;
Demod_SymbolMap_TypeDef SymbolMap[1024];
uint32_t MapNum = 0;
Demod_GenSymbolMap(ModType, SymbolMap, &MapNum);	

20. Pulse Detection (option)

20.1 Pulse_Open

int Pulse_Open(void Device)**	
Description	
Enable the pulse detection feature, check for the required license, and allocate memory needed to run the pulse detection.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **Device**	Device handle. After the function call, it will return the memory address used by the pulse detection feature, which serves as a reference for subsequent pulse detection function calls.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This should be called after Device_Open and must be called before calling any other pulse detection–related functions.
Example	Please refer to the relevant example of the Pulse_Detect() function.

20.2 Pulse_Close

int Pulse_Close(void Device)**	
Description	
Disable the pulse detection feature and release the memory previously allocated by Pulse_Open. Call this after all pulse detection operations are completed to free resources.	
Compatibility	0.55.55 and later.
Parameter description	
[in] **Device**	Device handle. Pointing to the memory address returned by Pulse_Open for the pulse detection feature.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Pulse_Open.
Example	Please refer to the relevant example of the Pulse_Detect() function.

20.3 Pulse_Detect

int Pulse_Detect(**void** Device,** **const Pulse_Profile_TypeDef* Pulse_Profile,** **PulseInfo_TypeDef* PulseInfo**)	
Description	
Perform pulse detection on DET data. Based on the input pulse data and threshold parameters, analyze and output pulse characteristics such as pulse width, period, and duty cycle. Must be called after Pulse_Open succeeds.	
Compatibility	0.55.55 and later.
Parameter description	
[in] Device	Device handle.
[in] Pulse_Profile	Input data for pulse detection, including the data to be analyzed and threshold parameters. Refer to the definition of the Pulse_Profile_TypeDef structure for details.
[out] PulseInfo	Output the pulse detection results, including pulse width, period, duty cycle, and other related parameters. Refer to the definition of the PulseInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Pulse_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); Status = Pulse_Open (&Device); DET_Profile_TypeDef DET_ProfileIn, DET_ProfileOut; DET_StreamInfo_TypeDef StreamInfo; DET_ProfileDeInit(&Device, &DET_ProfileIn); DET_ProfileIn.CenterFreq_Hz = 1e9; DET_ProfileIn.RefLevel_dBm = 0; DET_ProfileIn.DecimateFactor = 2; DET_ProfileIn.DecimateFactor = 1;	

```
DET_ProfileIn.TriggerMode = FixedPoints;
DET_ProfileIn.TriggerSource = Bus;
DET_ProfileIn.TriggerLength = 16242 * 10;
Status = DET_Configuration(&Device, &DET_ProfileIn, &DET_ProfileOut, &StreamInfo);
vector<float> NormalizedPowerStream(StreamInfo.PacketCount * StreamInfo.PacketSamples);
float ScaleToV = 0;
DET_TriggerInfo_TypeDef TriggerInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = DET_BusTriggerStart(&Device);
for (uint32_t i = 0; i < StreamInfo.PacketCount; i++) {
Status = DET_GetPowerStream(&Device, NormalizedPowerStream.data() + i *
StreamInfo.PacketSamples, &ScaleToV, &TriggerInfo, &MeasAuxInfo);
}
vector<float> NormalizedPowerStream_dBm(NormalizedPowerStream.size());
for (int i = 0; i < StreamInfo.StreamSamples; i++) {
NormalizedPowerStream_dBm[i] = 10 * log10(20 * pow(NormalizedPowerStream[i] * ScaleToV, 2));
}
Status = Pulse_Open(&Device);
Pulse_Profile_TypeDef Pulse_Profile;
Pulse_Profile.TimeResolution_s = StreamInfo.TimeResolution;
Pulse_Profile.PulseSize = NormalizedPowerStream_dBm.size();
Pulse_Profile.unit = Power_dBm;
Pulse_Profile.DetThreshold = -20;
Pulse_Profile.ExpPulseNum = 10;
Pulse_Profile.Pulse = NormalizedPowerStream_dBm.data();
PulseInfo_TypeDef PulseInfo;
Status = Pulse_Detect(&Device, &Pulse_Profile, &PulseInfo);
Status = Pulse_Close(&Device);
Status = Device_Close(&Device);
```

20.4 Pulse_Detect_PM1

int Pulse_Detect_PM1(**void** Device,** **const Pulse_Profile_TypeDef* Pulse_Profile,** **PulseInfoPM1_TypeDef* PulseInfoPM1** **)**	
Description	
Perform pulse detection on IQ data. Based on the input pulse data and threshold parameters, analyze and output pulse characteristics such as pulse width, period, duty cycle, pulse modulation type, pulse frequency, and phase information. Must be called after Pulse_Open succeeds.	
Compatibility	0.55.55 and later.
Parameter description	
[in] Device	Device handle.
[in] Pulse_Profile	Input data for pulse detection, including the data to be analyzed, threshold parameters, and other relevant settings. Refer to the definition of the Pulse_Profile_TypeDef structure for details.
[out] PulseInfoPM1	Output the pulse detection results, including pulse modulation type, pulse frequency, phase information, and other related parameters. Refer to the definition of the PulseInfoPM1_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Pulse_Open.
Example	
int Status = -1; int DeviceNum = 0; void* Device = NULL; BootProfile_TypeDef BootProfile; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; BootInfo_TypeDef BootInfo; Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo); Status = Pulse_Open(&Device); IQS_Profile_TypeDef ProfileIn, ProfileOut; IQS_StreamInfo_TypeDef StreamInfo; Status = IQS_ProfileDeInit(&Device, &ProfileIn); ProfileIn.CenterFreq_Hz = 1.00e9; ProfileIn.DecimateFactor = 2; ProfileIn.TriggerLength = 16242*100;	

```
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
void* AlternIQStream = NULL;
float ScaleToV = 0;
vector<float> IQ_Data(StreamInfo.StreamSamples * 2);
IQS_TriggerInfo_TypeDef TriggerInfo;
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = IQS_BusTriggerStart(&Device);
for (int j = 0; j < StreamInfo.PacketCount; j++) {
    Status = IQS_GetIQStream(&Device, &AlternIQStream, &ScaleToV, &TriggerInfo, &MeasAuxInfo);
    int16_t* IQ = (int16_t*)AlternIQStream;
    for (int i = 0; i < StreamInfo.PacketSamples * 2; i ++){
        IQ_Data[i + StreamInfo.PacketSamples * 2 * j] = (float)IQ[i] * ScaleToV;
    }
}
Status = IQS_BusTriggerStop(&Device);
Pulse_Profile_TypeDef Pulse_Profile;
Pulse_Profile.TimeResolution_s = 1.0 / StreamInfo.IQSampleRate;
Pulse_Profile.PulseSize = IQ_Data.size() / 2;
Pulse_Profile.unit = Power_dBm;
Pulse_Profile.DetThreshold = -20;
Pulse_Profile.ExpPulseNum = 10;
Pulse_Profile.Pulse = IQ_Data.data();
PulseInfoPM1_TypeDef PulseInfoPM1;
Status = Pulse_Detect_PM1(&Device, &Pulse_Profile, &PulseInfoPM1);
Status = Pulse_Close(&Device);
Status = Device_Close(&Device);
```

21. Auxiliary Signal Generator (option)

The ASG is an auxiliary signal source function that can output single-tone or swept-frequency signals. Currently, only the N45, N60, M60, and M80 models support this optional feature.

21.1 ASG_ProfileDeInit

int ASG_ProfileDeInit(void** Device, ASG_Profile_TypeDef* Profile)	
Description	
Initialize the auxiliary signal generator's configuration parameters to their default state. This clears or resets all settings in the Profile, ensuring subsequent configurations start from a standard initial state and are not affected by previous settings.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Device**	Device handle.
[out] **Profile**	Pointer to the ASG configuration structure. Refer to the definition of the ASG_Profile_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the ASG_Configuration() function.

21.2 ASG_Configuration

int ASG_Configuration(void** Device, ASG_Profile_TypeDef* ProfileIn, ASG_Profile_TypeDef* ProfileOut, ASG_Info_TypeDef* ASG_Info)	
Description	
Configure the operating state of the analog signal source. Based on the input Profile parameters, set the signal source's operating mode, frequency, power, scanning, and triggering settings, and return the actual applied configuration along with the current signal source status.	
Compatibility	0.55.0 and later.

Parameter description	
[in] **Device**	Device handle reference used to identify the currently opened device instance.
[in] **ProfileIn**	Input analog signal source configuration structure, used to set the operating parameters of the signal source. Refer to the definition of the ASG_Profile_TypeDef structure for details.
[out] **ProfileOut**	Output structure for the analog signal source's actual applied configuration, reflecting the ASG settings currently in effect on the device. Refer to the definition of the ASG_Profile_TypeDef structure for details.
[out] **ASG_Info**	ASG-related information in ASG mode. Refer to the definition of the ASG_Info_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
ASG_Profile_TypeDef ProfileIn, ProfileOut;
ASG_Info_TypeDef ASG_Info;
Status = ASG_ProfileDeInit(&Device, &ProfileIn);
ProfileIn.CenterFreq_Hz = 1e9;
Status = ASG_Configuration(&Device, &ProfileIn, &ProfileOut, &ASG_Info);
Status = Device_Close(&Device);
```

21.3 ASG_Configuration_WithExtMultiplier

```
int ASG_Configuration_WithExtMultiplier(
    void** Device,
    ASG_Profile_TypeDef* ProfileIn,
    ASG_Profile_TypeDef* ProfileOut,
    ASG_Info_TypeDef* ASG_Info,
    uint32_t Harmonic
)
```

Description

Configures the operating state of the analog signal generator. This function is primarily intended for applications where the tracking generator (TG) operates together with an external frequency multiplier.

Harmonic specifies the multiplication factor of the external frequency multiplier so that the actual ASG frequency range matches the external frequency multiplication relationship.	
Compatibility	0.55.99 and later.
Parameter description	
[in] **Device**	Device handle reference used to identify the currently opened device instance.
[in] **ProfileIn**	Input analog signal source configuration structure, used to set the operating parameters of the signal source. Refer to the definition of the ASG_Profile_TypeDef structure for details.
[out] **ProfileOut**	Output structure for the analog signal source's actual applied configuration, reflecting the ASG settings currently in effect on the device. Refer to the definition of the ASG_Profile_TypeDef structure for details.
[out] **ASG_Info**	ASG-related information in ASG mode. Refer to the definition of the ASG_Info_TypeDef structure for details.
[in] **Harmonic**	Multiplication factor of the external frequency multiplier.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
ASG_Profile_TypeDef asg, asg_fb;
ASG_Info_TypeDef asg_info;
Status = ASG_ProfileDeInit(device, &asg);
asg.Mode = ASG_TrackGenerator;
asg.Level_dBm = -10;
Status = ASG_Configuration_WithExtMultiplier(device, &asg, &asg_fb, &asg_info, 1);
Status = Device_Close(&Device);
```

22. EMI (option)

The EMI measurement option is used to implement the electromagnetic interference testing workflow, including frequency-band pre-scanning, abnormal signal searching, and precise spot-frequency measurement.

22.1 emi_config_scan

```
int emi_config_scan(
    void** device,
    const emi_scan_profile* scan_profile_i,
    emi_scan_profile* scan_profile_o,
    emi_scan_info* scan_info_o
)
```

Description	
Configure EMI pre-scan frequency band parameters.	
Compatibility	0.55.89 and later.
Parameter description	
[in] device	Device handle.
[in] scan_profile_i	Input pre-scan configuration parameters. Refer to the definition of the emi_scan_profile structure for details.
[out] scan_profile_o	Return the actual effective pre-scan configuration parameters. Refer to the definition of the emi_scan_profile structure for details.
[out] scan_info_o	Return the pre-scan trace information. Refer to the definition of the emi_scan_info structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.

Example	Please refer to the relevant example of the emi_get_partial_trace() function.

22.2 emi_get_partial_trace

```
int emi_get_partial_trace (
    void** device,
    double freq_o[],
    float amp_peak_o[],
    float amp_rms_o[],
    float amp_min_o[],
    float amp_sample_o[],
    int* index_o,
    MeasAuxInfo_TypeDef* aux_info_o,
    uint8_t* completed_o
)
```

Description	
Retrieve the EMI pre-scan segmented trace.	
Compatibility	0.55.89 and later.
Parameter description	
[in] **device**	Device handle.
[out] **freq_o**	Return the frequency axis, in Hz.
[out] **amp_peak_o**	Return the Peak power axis.
[out] **amp_rms_o**	Return the RMS power axis.
[out] **amp_min_o**	Return the Min power axis.
[out] **amp_sample_o**	Return the Sample power axis.

[out] **index_o**	Return the current segment index, used for stitching the complete trace.
[out] **aux_info_o**	Return auxiliary measurement information Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
[out] **completed_o**	Return the frequency band scan completion flag, where 1 indicates completed.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after emi_config_scan.
Example	

```
emi_scan_profile scan_i = {};
emi_scan_profile scan_o = {};
emi_scan_info scan_info = {};
scan_i.start = 300e6;
scan_i.stop = 1000e6;
scan_i.rbw = 120e3;
scan_i.vbw = 120e3;
scan_i.filter_dB = 6;
scan_i.unit = DBUV;
scan_i.ref_level = 97;
scan_i.dwell_time = 0.01f;
scan_i.detector = EMI_MAX_DETECTOR | EMI_RMS_DETECTOR;
scan_i.window = CISPR;
status = emi_config_scan(&device, &scan_i, &scan_o, &scan_info);
do {
    status = emi_get_partial_trace(&device, freq, peak, rms, min,
                                   sample, &index, &aux_info, &completed);
    // Stitch the current segment data based on index
} while (!completed);
```

22.3 emi_search

int emi_search (

```
    search_mode mode,
    double freq[],
    float amplitude[],
    uint32_t points,
    double limit_freq[],
    float limit_amplitude[],
    uint32_t limit_points,
    float margin,
    spur result_o[],
    uint32_t* signalnums_o
)
```

Description	
Compare the pre-scan trace with the EMI standard to search for anomalous signals. If the standard limit is a Peak or QP standard, please pass the Peak power axis. If the standard limit is an RMS or linAvg standard, please pass the RMS power axis.	
Compatibility	0.55.89 and later.
Parameter description	
[in] mode	Signal search mode. SPUR compares the trace against the specified limit line, while PEAK compares the trace against the local noise floor.
[in] freq	Trace frequency array.
[in] amplitude	Trace amplitude array.
[in] points	Number of trace points.
[in] limit_freq	Limit line frequency array in SPUR mode, ignored in PEAK mode.
[in] limit_amplitude	Limit line amplitude array in SPUR mode, ignored in PEAK mode (same unit as the trace).

[in] **limit_points**	Number of limit line points.
[in] **margin**	In SPUR mode, limit + margin is used as the search threshold (recommended range: -20 to 20 dB). In PEAK mode, noise floor + margin is used as the search threshold (recommended range: 6 to 42 dB).
[out] **result_o**	Detection results array, with a maximum capacity of 256.
[out] **signalnums_o**	Actual number of output results.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after emi_get_partial_trace().
Example	

limit_freq[] = {30e6, 230e6, 230e6, 300e6};
limit_amplitude[] = {40.0f, 40.0f, 47.0f, 47.0f};
spur search_results[EMI_MAX_SIGNALS] = {};
uint32_t result_count = 0;
// This example uses the QP limit of CISPR-16 Band C to compare with the stitched Peak trace
emi_search(SPUR, full_freq.data(), full_amp_peak.data(), scan_info_o.total_points, limit_freq, limit_amplitude, 4, 0.0f, search_results, &result_count);	

22.4 emi_config_measure

```
int emi_config_measure (
    void** device,
    const emi_measure_profile* meas_profile_i,
    emi_measure_profile* meas_profile_o
)
```

Description	
Configure EMI frequency point measurement parameters.	
Compatibility	0.55.89 and later.

Parameter description	
[in] device	Device handle.
[in] meas_profile_i	Input frequency point measurement configuration parameters. Refer to the definition of the emi_measure_profile structure for details.
[out] meas_profile_o	Return the actual effective frequency point measurement configuration parameters. Refer to the definition of the emi_measure_profile structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.
Example	Please refer to the relevant example of the emi_reset_measure() function.

22.5 emi_get_measure_result

```
int emi_config_scan(
    void** device,
    float* peak_o,
    float* rms_avg_o,
    float* lin_avg_o,
    float* quasi_peak_o,
    float* measure_time_o,
    MeasAuxInfo_TypeDef* aux_info_o,
    uint8_t* completed_o
)
```

Description
Retrieve EMI frequency point measurement results.

Compatibility	0.55.89 and later.
Parameter description	
[in] device	Device handle.
[out] peak_o	Return the peak detection result.
[out] rms_avg_o	Return the RMS detection result.
[out] lin_avg_o	Return the linear average detection result.
[out] quasi_peak_o	Return the quasi-peak detection result.
[out] measure_time_o	Return the elapsed measurement time, in s.
[out] aux_info_o	Return auxiliary measurement information.
[out] completed_o	Return the frequency point measurement completion flag, where 1 indicates completed.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after emi_start_measure.
Example	Please refer to the relevant example of the emi_reset_measure() function.

22.6 emi_start_measure

```
int emi_start_measure (
    void**device
)
```

Description	
Start EMI frequency point measurement.	
Compatibility	0.55.89 and later.
Parameter description	
[in] device	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.

Calling constraints	Must be called after emi_config_measure.
Example	Please refer to the relevant example of the emi_reset_measure() functiom.

22.7 emi_stop_measure

int emi_stop_measure (**void**device**)
Description
Stop EMI frequency point measurement.

Compatibility	0.55.89 and later.
Parameter description	
[in] **device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after emi_start_measure.
Example	Please refer to the relevant example of the emi_reset_measure() function.

22.8 emi_reset_measure

int emi_start_measure (**void**device**)
Description
Reset EMI frequency point measurement.

Compatibility	0.55.89 and later.
Parameter description	

[in] **device**	Device handle.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after Device_Open.

示例

```
emi_measure_profile meas_i = {};
emi_measure_profile meas_o = {};
meas_i.center = 1000e6;
meas_i.unit = DBUV;
meas_i.ref_level = 97;
meas_i.bw = 120e3;
meas_i.filter_dB = 6.0f;
meas_i.charge_time = 1e-3f;    // 1ms
meas_i.decay_time = 550e-3f; // 550ms
meas_i.mode = -1; // Automatically converge measurement results
meas_i.measure_time = 1.0f; // When mode = -1, this parameter is disabled
status = emi_config_measure(&device, &meas_i, &meas_o);
status = emi_start_measure(&device);
do {
      status = emi_get_measure_result(&device, &peak, &rms, &avg,
                                                  &qpeak, &meas_time, &aux_info, &completed);
} while (!completed);
status = emi_stop_measure(&device);
```

23. Analog Signal Demodulation Mode

ADM is an interface for analog modulation and demodulation processing, allowing users to call its functions to perform analog signal demodulation.

23.1 ADM_Open

void ADM_Open(void ADM)**	
Description	
Create and initialize an analog demodulation instance, allocating the necessary resources in memory. This function must be called before invoking any other analog demodulation–related functions.	
Compatibility	0.55.77 and later.
Parameter description	
[out] **ADM**	Memory reference for the analog demodulation feature. Upon successful call, it returns the address of the created ADM instance, which must be used for all subsequent analog demodulation–related API operations.
Return value	None.
Calling constraints	This function should be called before any other analog demodulation functions and only needs to be called once at the very beginning. Subsequent functions can operate using the device memory address returned by this function. For any normal ADM_Open call, ADM_Close must be called after all functionality is finished to release the allocated memory.
Example	Please refer to the relevant example of the ADM_AMDemod_PM1() function.

23.2 ADM_Close

void ADM_Close(void ADM)**	
Description	
Close and release the analog demodulation instance, freeing the internal resources and memory allocated by ADM_Open. This function must be called when the analog demodulation feature is no longer in use.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **ADM**	Memory reference for the analog demodulation feature, returned by ADM_Open. After calling this function, the instance becomes invalid and its internal resources are released.

Return value	None.
Calling constraints	This function only needs to be called at the end of program execution. After calling it, the analog demodulation feature is closed and the memory is released. To use the analog demodulation feature again, call ADM_Open to reopen the Analog functionality.
Example	Please refer to the relevant example of the ADM_AMDemod_PM1() function.

23.3 ADM_AMDemod

```
int ADM_AMDemod(
    void** ADM,
    const void* AlternIQStream,
    DataFormat_TypeDef DataFormat,
    uint64_t SamplePoints,
    double IQSampleRate,
    float DemodWaveform[]
)
```

Description	
Perform AM demodulation on the input IQ data, outputting only the demodulated time-domain waveform.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **ADM**	Reference to the analog demodulation instance, created by ADM_Open, used to identify the current ADM demodulation context.
[in] **AlternIQStream**	Input interleaved IQ data buffer, with the data format specified by DataFormat.
[in] **DataFormat**	Data format of the input IQ data, specifying the arrangement and precision of the IQ samples in memory. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
[in] **SamplePoints**	Number of sampling points in the input IQ data.
[in] **IQSampleRate**	Sampling rate of the input IQ data, in Hz.
[out] **DemodWaveform**	Array for the demodulated AM time-domain waveform, with a length corresponding to SamplePoints.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function only needs to be called at the end of program execution. After calling it, the analog demodulation feature is closed and its memory is released. To use the analog demodulation feature again, call ADM_Open to reopen the Analog functionality.
Example	Please refer to the relevant example of the ADM_AMDemod_PM1() function.

23.4 ADM_FMDemod

```
int ADM_FMDemod(
    void** ADM,
    const void* AlternIQStream,
    DataFormat_TypeDef DataFormat,
    uint64_t SamplePoints,
    double IQSampleRate,
    bool reset,
    float DemodWaveform[]
)
```

Description

Perform FM demodulation on the input IQ data, outputting only the demodulated time-domain waveform.

Compatibility	0.55.77 and later.
Parameter description	
[in] ADM	The instance for analog demodulation, created by ADM_Open, used to reference the current ADM demodulation context.
[in] AlternIQStream	The input interleaved IQ data buffer, with data format specified by DataFormat.
[in] DataFormat	The data format of the input IQ data, used to indicate the memory layout and precision of the IQ samples. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
[in] SamplePoints	The number of sample points in the input IQ data.
[in] IQSampleRate	The sampling rate of the input IQ data, in Hz.
[in] reset	Whether to reset the demodulator state. true clears the previous residual state before demodulation, while false preserves the accumulated state.
[out] DemodWaveform	The time-domain waveform array after FM demodulation, with length corresponding to SamplePoints.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after ADM_Open.
Example	Please refer to the relevant example of the ADM_FMDemod_PM1() function.

23.5 ADM_AMDemod_PM1

```
int ADM_AMDemod_PM1(
    void** ADM,
    const void* AlternIQStream,
    DataFormat_TypeDef DataFormat,
    uint64_t SamplePoints,
    double IQSampleRate,
    float IQS_ScaleToV,
    AMDemodParam_TypeDef* AMDemodParam
)
```

Description	
Performs AM demodulation on the input IQ data, computing and returning AM-related parameters while outputting the demodulated waveform, for analyzing the modulation characteristics of the AM signal.	
Compatibility	0.55.77 and later.
Parameter description	
[in] ADM	Reference to the analog demodulation instance, created by ADM_Open, used to identify the current ADM demodulation context.
[in] AlternIQStream	Returns the current device's GNSS satellite information, including the number of visible satellites and the satellites used for positioning.
[in] DataFormat	The data format of the input IQ data, indicating its memory layout and precision. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
[in] SamplePoints	The number of sample points in the input IQ data.
[in] IQSampleRate	The sampling rate of the input IQ data, in Hz.
[in] IQS_ScaleToV	The scale factor for converting IQ sample values to voltage, used for computing physical quantities of AM modulation parameters.
[out] AMDemodParam	Returns the modulation parameters and related analysis results after AM demodulation.Refer to the definition of the AMDemodParam_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after ADM_Open.
Example	

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
```

```
Device_Open_ErrorHandling(Status, &Device, DevNum, &BootProfile, &BootInfo);
IQStream_TypeDef IQStream;
IQS_Profile_TypeDef IQS_ProfileIn;
IQS_Profile_TypeDef IQS_ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &IQS_ProfileIn);
IQS_ProfileIn.CenterFreq_Hz = 1e9;
IQS_ProfileIn.RefLevel_dBm = 0;
IQS_ProfileIn.DataFormat = Complex16bit;
IQS_ProfileIn.TriggerMode = FixedPoints;
IQS_ProfileIn.TriggerSource = Bus;
IQS_ProfileIn.DecimateFactor = 256;
IQS_ProfileIn.BusTimeout_ms = 5000;
IQS_ProfileIn.TriggerLength = 16242 * 1;
Status = IQS_Configuration(&Device, &IQS_ProfileIn, &IQS_ProfileOut, &StreamInfo);
IQS_Configuration_ErrorHandling(Status, &Device, DevNum, &BootProfile, &BootInfo, &IQS_ProfileIn,
&IQS_ProfileOut, &StreamInfo);
AMDemodParam_TypeDef AMDemodParam; //Demod
void* ADM = nullptr;
ADM_Open(&ADM);
vector<int16_t> IQ(StreamInfo.StreamSamples * 2);
vector<float> DemodWaveform(IQS_ProfileIn.TriggerLength);
while(1) {
uint32_t Points = StreamInfo.PacketSamples;
Status = IQS_BusTriggerStart(&Device);
for (uint32_t i = 0; i < StreamInfo.PacketCount; i++) {
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
if (i == StreamInfo.PacketCount - 1 && StreamInfo.StreamSamples % StreamInfo.PacketSamples != 0) {
Points = StreamInfo.StreamSamples % StreamInfo.PacketSamples;
}
memcpy(IQ.data() + i * StreamInfo.PacketSamples * 2, IQStream.AlternIQStream, Points * 2 *
sizeof(IQ[0]));
}
IQStream.AlternIQStream = IQ.data();
Status = ADM_AMDemod(&ADM, IQStream.AlternIQStream, IQStream.IQS_Profile.DataFormat,
IQStream.IQS_Profile.TriggerLength, IQStream.IQS_StreamInfo.IQSampleRate,
DemodWaveform.data());
Status = ADM_AMDemod_PM1(&ADM, IQStream.AlternIQStream, IQStream.IQS_Profile.DataFormat,
IQStream.IQS_Profile.TriggerLength, IQStream.IQS_StreamInfo.IQSampleRate,
IQStream.IQS_ScaleToV, &AMDemodParam);
}
ADM_Close(&ADM);
Status = Device_Close(&Device);
```

23.6 ADM_FMDemod_PM1

```
int ADM_FMDemod_PM1(
    void** ADM,
    const void* AlternIQStream,
    DataFormat_TypeDef DataFormat,
    uint64_t SamplePoints,
    double IQSampleRate,
    float IQS_ScaleToV,
    bool reset,
    FMDemodParam_TypeDef* FMDemodParam
)
```

Description	
Performs FM demodulation on the input IQ data, returning both the demodulated waveform and modulation parameters for analyzing the FM signal's modulation characteristics. Optionally resets the demodulator state before demodulation.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **ADM**	Reference to the analog demodulation instance, created by ADM_Open, used to identify the current ADM demodulation context.
[in] **AlternIQStream**	The input interleaved IQ data buffer, with its format specified by DataFormat.
[in] **DataFormat**	The data format of the input IQ data, used to indicate its memory layout and precision. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
[in] **SamplePoints**	The number of sample points in the input IQ data.
[in] **IQSampleRate**	The sampling rate of the input IQ data, in Hz.
[in] **IQS_ScaleToV**	The scale factor for converting IQ sample values to voltage, used for calculating the physical quantities of AM modulation parameters.
[in] **reset**	Resets the buffer. For a single continuous demodulation session, set to 1 only on the first function call; use 0 for all subsequent calls.
[out] **FMDemodParam**	Returns the modulation parameters and related analysis results after FM demodulation. Refer to the definition of the FMDemodParam_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after ADM_Open.

Example

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
IQStream_TypeDef IQStream;
IQS_Profile_TypeDef IQS_ProfileIn;
IQS_Profile_TypeDef IQS_ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &IQS_ProfileIn);
IQS_ProfileIn.CenterFreq_Hz = 1e9;
IQS_ProfileIn.RefLevel_dBm = 0;
IQS_ProfileIn.DataFormat = Complex16bit;
IQS_ProfileIn.TriggerMode = FixedPoints;
IQS_ProfileIn.TriggerSource = Bus;
IQS_ProfileIn.DecimateFactor = 256;
IQS_ProfileIn.BusTimeout_ms = 5000;
IQS_ProfileIn.TriggerLength = 16242 * 1;
Status = IQS_Configuration(&Device, &IQS_ProfileIn, &IQS_ProfileOut, &StreamInfo);
FMDemodParam_TypeDef FMDemodParam; //Demod
void* ADM = nullptr;
ADM_Open(&ADM);
vector<int16_t> IQ(StreamInfo.StreamSamples * 2);
vector<float> DemodWaveform(IQS_ProfileIn.TriggerLength);
while(1) {
uint32_t Points = StreamInfo.PacketSamples;
Status = IQS_BusTriggerStart(&Device);
for (uint32_t i = 0; i < StreamInfo.PacketCount; i++) {
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
if (i == StreamInfo.PacketCount - 1 && StreamInfo.StreamSamples % StreamInfo.PacketSamples != 0) {
Points = StreamInfo.StreamSamples % StreamInfo.PacketSamples;
}
memcpy(IQ.data() + i * StreamInfo.PacketSamples * 2, IQStream.AlternIQStream, Points * 2 *
sizeof(IQ[0]));
}
IQStream.AlternIQStream = IQ.data();
```

```
Status = ADM_FMDemod(&ADM, IQStream.AlternIQStream, IQStream.IQS_Profile.DataFormat,
IQStream.IQS_Profile.TriggerLength, IQStream.IQS_StreamInfo.IQSampleRate,
true,DemodWaveform.data());
Status = ADM_FMDemod_PM1(&ADM, IQStream.AlternIQStream, IQStream.IQS_Profile.DataFormat,
IQStream.IQS_Profile.TriggerLength, IQStream.IQS_StreamInfo.IQSampleRate,
IQStream.IQS_ScaleToV, true, &FMDemodParam);
}
ADM_Close(&ADM);
Status = Device_Close(&Device);
```

24. Digital Signal Processing (Trace analysis)

24.1 DSP_TraceAnalysis_IM3

int DSP_TraceAnalysis_IM3(**const double Freq_Hz[],** **const float PowerSpec_dBm[],** **const uint32_t TracePoints,** TraceAnalysisResult_IP3_TypeDef*** IM3Result**)	
Description	
Performs IM3 parameter analysis on the input spectrum trace data, calculating and outputting third-order intermodulation results based on the frequency and power spectrum data.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[out] **IM3Result**	Returns the third-order intermodulation (IM3) analysis results, including the fundamental signal frequencies and powers, intermodulation products, and the IP3 calculation results. Refer to the definition of the TraceAnalysisResult_IP3_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.
Example	
int Status = 0; void* Device = NULL; int DevNum = 0; BootProfile_TypeDef BootProfile; BootInfo_TypeDef BootInfo; BootProfile.DevicePowerSupply = USBPortAndPowerPort; BootProfile.PhysicalInterface = USB; Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo); SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut; SWP_TraceInfo_TypeDef TraceInfo; Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);	

```
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
Status = SWP_AutoSet(&Device, SWPIM3Meas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.Span_Hz = 10e6;
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
TraceAnalysisResult_IP3_TypeDef    IM3Result;
Status = DSP_TraceAnalysis_IM3(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, &IM3Result);
Status = Device_Close(&Device);
```

24.2 DSP_TraceAnalysis_IM2

```
int DSP_TraceAnalysis_ IM2(
    const double Freq_Hz[],
    const float PowerSpec_dBm[],
    uint32_t TracePoints,
    TraceAnalysisResult_IP2_TypeDef* IM2Result
)
```

Description

Performs IM2 parameter analysis on the input spectrum trace data, calculating and outputting second-order intermodulation results based on the frequency and power spectrum data.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[out] **IM2Result**	Returns the second-order intermodulation (IM2) analysis results, including the fundamental signal frequencies and powers, second-order intermodulation products, and the IP2 calculation results. Refer to the definition of the TraceAnalysisResult_IP2_TypeDef structure for details.

Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.

Example

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
Status = SWP_AutoSet(&Device, SWPIM3Meas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.Span_Hz = 10e6;
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
TraceAnalysisResult_IP2_TypeDef IM2Result;
Status = DSP_TraceAnalysis_IM2(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, &IM2Result);
Status = Device_Close(&Device);
```

24.3 DSP_TraceAnalysis_ChannelPower

```
int DSP_TraceAnalysis_ChannelPower(
    const double Freq_Hz[],
    const float PowerSpec_dBm[],
    const uint32_t TracePoints,
    const double CenterFrequency,
    const double AnalysisSpan,
    const double RBW,
    DSP_ChannelPowerInfo_TypeDef* ChannelPowerResult
)
```

Description

Performs channel power analysis on the input spectrum trace data, calculating the power within the specified channel based on the center frequency, analysis bandwidth, and resolution bandwidth, and outputs the analysis results.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[in] **CenterFrequency**	The center frequency of the channel to be measured.
[in] **AnalysisSpan**	The bandwidth of the channel to be measured.
[in] **RBW**	The resolution bandwidth used for the analysis.
[out] **ChannelPowerResult**	Returns the channel power analysis results, including the total channel power, power density, and the frequency, power, and index of the peak within the channel. Refer to the definition of the DSP_ChannelPowerInfo_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
SWP_AutoSet(&Device, SWPChannelPowerMeas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.Span_Hz = 10e6;
```

```
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
double CenterFrequency = 1e9;
double AnalysisSpan = 2e6;
DSP_ChannelPowerInfo_TypeDef ChannelPowerResult;
Status = DSP_TraceAnalysis_ChannelPower(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, CenterFrequency, AnalysisSpan, SWP_ProfileOut.RBW_Hz,
&ChannelPowerResult);
Status = Device_Close(&Device);
```

24.4 DSP_TraceAnalysis_XdBBW

```
int DSP_TraceAnalysis_XdBBW(
    const double Freq_Hz[],
    const float PowerSpec_dBm[],
    const uint32_t TracePoints,
    const float XdB,
    TraceAnalysisResult_XdB_TypeDef* XdBResult
)
```

Description

Performs XdB bandwidth analysis on the input spectrum trace data, calculating the signal bandwidth relative to the peak power drop of XdB, and outputs the analysis results.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[in] **XdB**	The power drop relative to the peak, used to define the XdB bandwidth.
[out] **XdBResult**	Returns the XdB bandwidth analysis results, including the bandwidth, center frequency, start and stop frequencies, and the frequency, power, and index of the peak within the bandwidth. Refer to the definition of the TraceAnalysisResult_XdB_TypeDef structure for details.

Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
Status = SWP_AutoSet(&Device, SWPOBWMeas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.RBW_Hz = 50e3;
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
float XdB = 3;
TraceAnalysisResult_XdB_TypeDef XdBResult;
Status = DSP_TraceAnalysis_XdBBW(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, XdB, &XdBResult);
Status = Device_Close(&Device);
```

24.5 DSP_TraceAnalysis_OBW

```
int DSP_TraceAnalysis_OBW(
    const double Freq_Hz[],
    const float PowerSpec_dBm[],
    const uint32_t TracePoints,
    const float OccupiedRatio,
    TraceAnalysisResult_OBW_TypeDef* OBWResult
)
```

Description

Performs occupied bandwidth analysis on the input spectrum trace data, calculating the signal bandwidth corresponding to a specified power percentage, and outputs the analysis results.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[in] **OccupiedRatio**	The occupied bandwidth ratio to be measured, typically set to 0.99.
[out] **OBWResult**	Returns the occupied bandwidth analysis results, including the occupied bandwidth, center frequency, start and stop frequencies with corresponding power and power ratio, and the frequency, power, and index of the peak within the bandwidth. Refer to the definition of the TraceAnalysisResult_OBW_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
Status = SWP_AutoSet(&Device, SWPOBWMeas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.Span_Hz = 10e6;
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
```

```
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
float OccupiedRatio = 0.99;
TraceAnalysisResult_OBW_TypeDef OBWResult;
Status = DSP_TraceAnalysis_OBW(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, OccupiedRatio, &OBWResult);
Status = Device_Close(&Device);
```

24.6 DSP_TraceAnalysis_ACPR

```
int DSP_TraceAnalysis_ACPR(
    const double Freq_Hz[],
    const float PowerSpec_dBm[],
    const uint32_t TracePoints,
    const DSP_ACPRFreqInfo_TypeDef ACPRFreqInfo,
    TraceAnalysisResult_ACPR_TypeDef* ACPRResult
)
```

Description

Performs adjacent channel power ratio (ACPR) analysis on the input spectrum trace data, calculating the power ratio between the main channel and adjacent channels based on the specified adjacent channel frequency information, and outputs the analysis results.

Compatibility	0.55.77 and later.
Parameter description	
[in] **Freq_Hz[]**	The input frequency array, corresponding to the frequency values of each point in the trace, in Hz.
[in] **PowerSpec_dBm[]**	The input power array, corresponding to the power values of each point in the trace, in dBm.
[in] **TracePoints**	The number of points in the trace, representing the length of the frequency and power arrays.
[in] **ACPRFreqInfo**	The frequency information required for adjacent channel power ratio (ACPR) analysis, including the resolution bandwidth, main channel center frequency and bandwidth, adjacent channel spacing, and the number of adjacent channel pairs. Refer to the definition of the DSP_ACPRFreqInfo_TypeDef structure for details.
[out] **ACPRResult**	Returns the adjacent channel power ratio (ACPR) analysis results, including the main channel power and peak information, as well as the center frequency, bandwidth, power, power ratio, power difference, and peak information of the left and right adjacent channels.

	Refer to the definition of the TraceAnalysisResult_ACPR_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after SWP_Get related-function.
Example	

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
SWP_Profile_TypeDef SWP_ProfileIn, ProfileSetOut, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
Status = SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.CenterFreq_Hz = 1e9;
SWP_ProfileIn.FreqAssignment = CenterSpan;
uint8_t IfDoConfig = 1;
Status = SWP_AutoSet(&Device, SWPACPRMeas, &SWP_ProfileIn, &ProfileSetOut, &TraceInfo,
IfDoConfig);
ProfileSetOut.Span_Hz = 10e6;
Status = SWP_Configuration(&Device, &ProfileSetOut, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
MeasAuxInfo_TypeDef MeasAuxInfo;
Status = SWP_GetFullSweep(&Device, Frequency.data(), PowerSpec_dBm.data(), &MeasAuxInfo);
DSP_ACPRFreqInfo_TypeDef ACPRFreqInfo;
ACPRFreqInfo.RBW = SWP_ProfileOut.RBW_Hz;
ACPRFreqInfo.AdjChPair = 2;
ACPRFreqInfo.AdjChSpace_Hz = 2e6;
ACPRFreqInfo.MainChBW_Hz = 1e6;
ACPRFreqInfo.MainChCenterFreq_Hz = 1e9;
vector<TraceAnalysisResult_ACPR_TypeDef> ACPRResult(ACPRFreqInfo.AdjChPair);
Status = DSP_TraceAnalysis_ACPR(Frequency.data(), PowerSpec_dBm.data(),
TraceInfo.FullsweepTracePoints, ACPRFreqInfo, ACPRResult.data());
Status = Device_Close(&Device);
```

25. Digital Signal Processing (Processing of stream)

25.1 DSP_Open

int DSP_Open(void DSP)**	
Description	
Opens the DSP function and allocates memory for storing DSP-related data. This function must be called before invoking any other DSP functions. Multiple DSP instances can be operated simultaneously by providing multiple DSP pointers.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	Reference to the memory space required for running the DSP. After calling, it returns the memory address of the currently opened DSP instance, which must be used to index this address in subsequent API calls.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before any other DSP function calls and only once at the start of the program. After use, DSP_Close must be called to release the allocated memory.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.2 DSP_Close

int DSP_ Close(void DSP)**	
Description	
Closes the DSP function and releases the memory previously allocated by DSP_Open.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	Reference to the memory space required for running the DSP, pointing to the memory address returned by DSP_Open, used for subsequent DSP operations or memory release.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	This function must be called at the end of program execution. After calling, the DSP function is closed and the memory is released. To use the DSP function again, DSP_Open must be called.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.3 DSP_FFT_DeInit

int DSP_FFT_DeInit(DSP_FFT_TypeDef* IQToSpectrum)	
Description	
Initializes the FFT mode parameters, including FFT size, window type, decimation factor, etc., with all parameters encapsulated in the DSP_FFT_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[out] **IQToSpectrum**	Configures the FFT mode parameter structure, including the number of analysis points, number of valid sample points, window type, trace detection method, detection ratio, spectrum cropping ratio, and whether calibration is applied. Refer to the definition of the DSP_FFT_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before DSP_FFT_Configuration.
Example	Please refer to the relevant example of the DSP_FFT_IQSToSpectrum() function.

25.4 DSP_FFT_Configuration

int DSP_FFT_Configuration(void** DSP, const DSP_FFT_TypeDef* ProfileIn, DSP_FFT_TypeDef* ProfileOut, uint32_t* TracePoints, double* RBWRatio)	
Description	
Configures the relevant parameters for FFT mode. Parameters such as FFT size, window type, and decimation factor in FFT mode are all encapsulated in the DSP_FFT_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	DSP memory reference returned by DSP_Open, used to identify the current DSP instance.
[in] **ProfileIn**	The input FFT configuration parameters, including the number of analysis points, number of sampling points, window type, trace detection method, and others. Refer to the definition of the DSP_FFT_TypeDef structure for details.

[out] **ProfileOut**	Returns the FFT configuration parameters actually applied, which can be used to verify the configuration. Refer to the definition of the DSP_FFT_TypeDef structure for details.
[out] **TracePoints**	The number of spectrum points available under the current DSP_FFT configuration.
[out] **RBWRatio**	Returns the ratio of the resolution bandwidth to the sampling rate. Calculation Formula: RBW = RBWRatio * IQSampleRate
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DSP_FFT_DeInit.
Example	Please refer to the relevant example of the DSP_FFT_IQSToSpectrum() function.

25.5 DSP_FFT_IQSToSpectrum

```
int DSP_FFT_IQSToSpectrum(
    void** DSP,
    const IQStream_TypeDef* IQStream,
    double Freq_Hz[],
    float PowerSpec_dBm[]
)
```

Description	
Converts the input IQ data stream into spectrum data, outputting the corresponding frequency and power arrays for spectrum analysis.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	DSP memory reference returned by DSP_Open, used to identify the current DSP instance.
[in] **IQStream**	Information related to the IQ data stream, including the IQ data and associated configuration parameters. Refer to the definition of the IQStream_TypeDef structure for details.
[out] **Freq_Hz[]**	Returns the frequency array of the spectrum data, in Hz. The array length is TracePoints, as determined by the DSP_FFT_Configuration() function.
[out] **PowerSpec_dBm[]**	Returns the power array of the spectrum data, in dBm. The array length is TracePoints, as determined by the DSP_FFT_Configuration() function.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DSP_FFT_Configuration.
Example	

```
void* Device = NULL; int DeviceNum = 0; int Status = -1;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
```

```
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
IQStream_TypeDef IQStream;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
vector<int16_t> AlternIQStream(StreamInfo.StreamSamples * 2);
void* DSP = NULL; uint32_t TracePoints = 0; double RBWRatio = 0;
Status = DSP_Open(&DSP);
DSP_FFT_TypeDef FFT_ProfileIn, FFT_ProfileOut;
Status = DSP_FFT_DeInit(&FFT_ProfileIn);
Status = DSP_FFT_Configuration(&DSP, &FFT_ProfileIn, &FFT_ProfileOut, &TracePoints, &RBWRatio);
vector<double> Frequency(TracePoints); vector<float> PowerSpec_dBm(TracePoints);
Status = IQS_BusTriggerStart(&Device);
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
Status = DSP_FFT_IQSToSpectrum(&DSP, &IQStream, Frequency.data(), PowerSpec_dBm.data());
Status = IQS_BusTriggerStop(&Device);
Status = DSP_Close(&DSP);
Status = Device_Close(&Device);
```

25.6 DSP_DDC_DeInit

int DSP_DDC_DeInit(DSP_DDC_TypeDef* DDC_ProfileIn)	
Description	
Initializes the parameters for DDC mode. In DDC mode, the complex mixing and resampling parameters are encapsulated in the DSP_DDC_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[out] **DDC_ProfileIn**	The input Digital Down Conversion (DDC) configuration parameters, including the complex mixing offset frequency, sampling rate, resampling decimation factor, and number of sample points, used to initialize or configure the DDC function. Refer to the definition of the DSP_DDC_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called before DSP_DDC_Configuration.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.7 DSP_DDC_Configuration

int DSP_DDC_Configuration(void** DSP, const DSP_DDC_TypeDef* DDC_ProfileIn, DSP_DDC_TypeDef* DDC_ProfileOut)	
Description	
Configure the parameters for DDC mode. In DDC mode, the complex mixing and resampling parameters are encapsulated in the DSP_DDC_TypeDef structure.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	The DSP memory reference returned by DSP_Open, used to index the current DSP instance.
[in] **ProfileIn**	The input DDC configuration parameters, including the complex mixing frequency offset, sampling rate, decimation factor, and number of sample points, used to configure the DDC function. Refer to the definition of the DSP_DDC_TypeDef structure for details.
[out] **ProfileOut**	Returns the DDC configuration parameters actually applied, which can be used to verify the configuration. Refer to the definition of the DSP_DDC_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DSP_DDC_DeInit.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.8 DSP_DDC_Reset

void DSP_DDC_Reset(void** DSP)	
Description	
Resets the buffer in the DDC function, clearing historical data to prepare for a new downconversion operation.	
Compatibility	0.55.77 and later.
Parameter description	
[in/out] **DSP**	DSP memory reference, returned by DSP_Open, used to index the current DSP instance.
Return value	None.
Calling constraints	Must be called after DSP_Open.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.9 DSP_DDC_GetDelay

void DSP_DDC_GetDelay(void** DSP, uint32_t* delay)	
Description	
Gets the current DDC processing latency. By providing the DSP memory reference, it returns the DDC latency under the current configuration (in number of samples), which can be used for calibration or synchronized processing.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	DSP memory reference, returned by DSP_Open, used to index the current DSP instance.
[out] **delay**	Returns the DDC processing latency, measured in number of samples.
Return value	None.
Calling constraints	Must be called after DSP_DDC_Configuration.
Example	Please refer to the relevant example of the DSP_DDC_Execute() function.

25.10 DSP_DDC_Execute

int DSP_DDC_Execute(void** DSP, const IQStream_TypeDef* IQStreamIn, IQStream_TypeDef* IQStreamOut)	
Description	
Performs Digital Down Conversion (DDC) operation, converting the input IQ data stream into a down-converted output IQ data stream based on the current DDC configuration, for subsequent processing or analysis.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	DSP memory reference, returned by DSP_Open, used to index the current DSP instance.
[in] **IQStreamIn**	Information regarding the input IQ data stream, including IQ data and associated configuration information. Refer to the definition of the IQStream_TypeDef structure for details.

[out] **IQStreamOut**	Information regarding the output IQ data stream, including IQ data and associated configuration information. Refer to the definition of the IQStream_TypeDef structure for details.
Return value	0: NoError. Nonzero: abnormal exist, please refer to the Appendix 1.
Calling constraints	Must be called after DSP_DDC_Configuration.
Example	

```
int Status = -1; void* DSP = NULL; void* Device = NULL; int DeviceNum = 0;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn;
IQS_Profile_TypeDef ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
IQStream_TypeDef IQStream;
IQStream_TypeDef IQStreamOut;
Status = IQS_BusTriggerStart(&Device);
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
Status = IQS_BusTriggerStop(&Device);
Status = DSP_Open(&DSP);
DSP_DDC_TypeDef DDC_ProfileIn, DDC_ProfileOut;
Status = DSP_DDC_DeInit(&DDC_ProfileIn);
uint32_t DDC_Points = 0;
Status = DSP_DDC_Configuration(&DSP, &DDC_ProfileIn, &DDC_ProfileOut);
uint32_t delay;
DSP_DDC_GetDelay(&DSP, &delay);
DSP_DDC_Reset(&DSP);
Status = DSP_DDC_Execute(&DSP, &IQStream, &IQStreamOut);
Status = DSP_Close(&DSP);
Status = Device_Close(&Device);
```

25.11 DSP_AudioAnalysis

void DSP_AudioAnalysis(**const double Audio[],** **const uint64_t SamplePoints,** **const double SampleRate,** **DSP_AudioAnalysis_TypeDef* AudioAnalysis**)
Description
Analyzes the input audio data, calculating and outputting key audio performance parameters, including audio voltage (V), audio frequency (Hz), SINAD (dB), and Total Harmonic Distortion (THD, %).

Compatibility	0.55.77 and later.
Parameter description	
[in] **Audio[]**	Input audio data array, containing audio sample values stored in chronological order.
[in] **SamplePoints**	Number of samples in the input audio data, representing the length of the Audio array.
[in] **SampleRate**	Sampling rate of the input audio data, measured in Hz.
[out] **AudioAnalysis**	Pointer to the structure returning audio analysis results, including audio voltage, audio frequency, SINAD, and THD. Refer to the definition of the DSP_AudioAnalysis_TypeDef structure for details.
Return value	None.
Calling constraints	Must be called after AM/FM demodulation function.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn, ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
IQStream_TypeDef IQStream;
void* AnalogMod = NULL;
ASD_Open(&AnalogMod);
```

```
bool reset = 1;
vector<float> result(StreamInfo.PacketSamples);
FM_DemodParam_TypeDef FM_DemodParam;
void* DSP = NULL;
DSP_Open(&DSP);
Status = IQS_BusTriggerStart(&Device);
DSP_AudioAnalysis_TypeDef    AudioAnalysis;
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
Status = ASD_FMDemodulation(&AnalogMod, &IQStream, reset, result.data(), &FM_DemodParam);
vector<double> Audio(result.begin(), result.end());
DSP_AudioAnalysis(Audio.data(), StreamInfo.PacketSamples, StreamInfo.IQSampleRate,
&AudioAnalysis);
Status = IQS_BusTriggerStop(&Device);
DSP_Close(&DSP);
ASD_Close(&AnalogMod);
Status = Device_Close(&Device);
```

25.12 DSP_LPF_DeInit

void DSP_LPF_DeInit(Filter_TypeDef* LPF_ProfileIn)	
Description	
Initializes or resets Low Pass Filter (LPF) parameters, preparing for subsequent filtering processing, including settings such as cutoff frequency and stopband attenuation.	
Compatibility	0.55.77 and later.
Parameter description	
[out] **LPF_ProfileIn**	Low-pass filter parameter structure, used for initializing or resetting filter configuration. Refer to the definition of the Filter_TypeDef structure for details.
Return value	None.
Calling constraints	Must be called before DSP_LPF_Configuration.
Example	Please refer to the relevant example of the DSP_LPF_Execute_Real() function.

25.13 DSP_LPF_Configuration

void DSP_LPF_Configuration(**void** DSP,** **const Filter_TypeDef* LPF_ProfileIn,** **Filter_TypeDef* LPF_ProfileOut** **)**	
Description	
Configures Low Pass Filter (LPF) parameters, generating and applying corresponding filter settings based on the input configuration, and returning the actual effective parameters for subsequent DSP filtering processing.	
Compatibility	0.55.77 and later.
Parameter description	
[in] DSP	DSP memory reference, returned by DSP_Open, used to index the current DSP instance.
[in] LPF_ProfileIn	Input low-pass filter configuration parameters, used to generate filter coefficients. Refer to the definition of the Filter_TypeDef structure for details.
[out] LPF_ProfileOut	Output actual low-pass filter configuration parameters, reflecting the LPF settings currently effective in the DSP. Refer to the definition of the Filter_TypeDef structure for details.
Return value	None.
Calling constraints	Must be called after DSP_LPF_DeInit.
Example	Please refer to the relevant example of the DSP_LPF_Execute_Real() function.

25.14 DSP_LPF_Reset

void DSP_LPF_Reset(void DSP)**	
Description	
Reset internal buffers and states of the Low Pass Filter (LPF) to clear historical filtering data, preventing influence on subsequent filtering results; typically called before restarting data processing or switching input data streams.	
Compatibility	0.55.77 and later.
Parameter description	
[in] DSP	DSP memory space reference, returned by DSP_Open, used to index the current DSP instance and access its LPF module.
Return value	None.
Calling constraints	Must be called after DSP_Open.
Example	Please refer to the relevant example of the DSP_LPF_Execute_Complex() function.

25.15 DSP_LPF_Execute_Real

void DSP_LPF_Execute_Real(**void** DSP,** **float Signal[],** **float LPF_Signal[]** **)**	
Description	
Apply low-pass filtering to a real-valued signal using the currently configured low-pass filter parameters, and output the resulting filtered real-valued signal.	
Compatibility	0.55.77 and later.
Parameter description	
[in] DSP	DSP memory handle returned by DSP_Open, used to reference the current DSP instance.
[in] Signal[]	Input array of real-valued signal data to be processed by the low-pass filter.
[out] LPF_Signal[]	Output array of real-valued signal data after low-pass filtering, with the same length as the input signal.
Return value	None.
Calling constraints	Must be called after DSP_LPF_Configuration.
Example	

int Status = -1; Sin_TypeDef NCO_Profile;
NCO_Profile.Amplitude = 10;
NCO_Profile.Frequency = 60e3;
NCO_Profile.Phase = 0;
NCO_Profile.SampleRate = 100e3;
NCO_Profile.Samples = 2000;
vector<float> sin(NCO_Profile.Samples);
vector<float> LPF_Signal(NCO_Profile.Samples);
void* DSP = NULL;
Status = DSP_Open(&DSP);
Filter_TypeDef LPF_ProfileIn;
Filter_TypeDef LPF_ProfileOut;
DSP_LPF_DeInit(&LPF_ProfileIn);
LPF_ProfileIn.As = 90;
LPF_ProfileIn.fc = 0.25;
LPF_ProfileIn.mu = 0;
LPF_ProfileIn.n = 90;	

```
LPF_ProfileIn.Samples = NCO_Profile.Samples;
DSP_LPF_Configuration(&DSP, &LPF_ProfileIn, &LPF_ProfileOut);
DSP_GenerateSineWaveform(sin.data(),&NCO_Profile);
DSP_LPF_Execute_Real(&DSP, sin.data(), LPF_Signal.data());
Status = DSP_Close(&DSP);
```

25.16 DSP_LPF_Execute_Complex

```
void DSP_LPF_Execute_Complex(
        void** DSP,
        const IQStream_TypeDef* IQStreamIn,
        IQStream_TypeDef* IQStreamOut
)
```

Description

Perform low-pass filtering on a complex IQ signal using the currently configured low-pass filter parameters, and output the resulting filtered complex IQ data stream.

Compatibility	0.55.77 and later.
Parameter description	
[in] **DSP**	DSP memory handle returned by DSP_Open, used to reference the current DSP instance and its low-pass filter configuration.
[in] **IQStreamIn**	Input complex IQ data stream to be processed by the low-pass filter. Refer to the definition of the IQStream_TypeDef structure for details.
[out] **IQStreamOut**	Output complex IQ data stream after low-pass filtering. Refer to the definition of the IQStream_TypeDef structure for details.
Return value	None.
Calling constraints	Must be called after DSP_LPF_Configuration.

Example

```
int Status = -1; int DeviceNum = 0; void* Device = NULL;
BootProfile_TypeDef BootProfile;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
BootInfo_TypeDef BootInfo;
Status = Device_Open(&Device, DeviceNum, &BootProfile, &BootInfo);
IQS_Profile_TypeDef ProfileIn;
IQS_Profile_TypeDef ProfileOut;
IQS_StreamInfo_TypeDef StreamInfo;
Status = IQS_ProfileDeInit(&Device, &ProfileIn);
Status = IQS_Configuration(&Device, &ProfileIn, &ProfileOut, &StreamInfo);
IQStream_TypeDef IQStream, IQStreamOut;
```

```
Status = IQS_BusTriggerStart(&Device);
Status = IQS_GetIQStream_PM1(&Device, &IQStream);
Status = IQS_BusTriggerStop(&Device);
void* DSP = NULL;
Status = DSP_Open(&DSP);
Filter_TypeDef LPF_ProfileIn, LPF_ProfileOut;
DSP_LPF_DeInit(&LPF_ProfileIn);
DSP_LPF_Configuration(&DSP, &LPF_ProfileIn, &LPF_ProfileOut);
DSP_LPF_Reset(&DSP);
DSP_LPF_Execute_Complex(&DSP, &IQStream, &IQStreamOut);
Status = DSP_Close(&DSP);
Status = Device_Close(&Device);
```

25.17 DSP_InterceptSpectrum

```
void DSP_InterceptSpectrum(
        const double StartFreq_Hz,
        const double StopFreq_Hz,
        const double Freq_Hz[],
        const float PowerSpec_dBm[],
        const uint32_t FullsweepTracePoints,
        double FrequencyOut[],
        float PowerSpecOut_dBm[],
        uint32_t* InterceptPoints
)
```

Description	
Intercept the redundant spectrum on both sides in SWP mode.	
Compatibility	0.55.77 and later.
Parameter description	
[in] **StartFreq_Hz**	Specify the start frequency of the spectrum segment to be intercepted, in Hz.
[in] **StopFreq_Hz**	Specify the stop frequency of the spectrum segment to be intercepted, in Hz.
[in] **Freq_Hz[]**	Array of frequencies to be intercepted, in Hz.
[in] **PowerSpec_dBm[]**	Array of power values to be intercepted, in dBm.
[in] **FullsweepTracePoints**	Number of trace points before interception.
[out] **FrequencyOut[]**	Array of intercepted frequencies, in Hz.
[out] **PowerSpecOut_dBm[]**	Array of intercepted power values, in dBm.
[out] **InterceptPoints**	Number of valid trace points actually intercepted.

Return value	None.
Calling constraints	None.
Example	

```
int Status = 0; void* Device = NULL; int DevNum = 0;
BootProfile_TypeDef BootProfile;
BootInfo_TypeDef BootInfo;
BootProfile.DevicePowerSupply = USBPortAndPowerPort;
BootProfile.PhysicalInterface = USB;
Status = Device_Open(&Device, DevNum, &BootProfile, &BootInfo);
DeviceInfo_TypeDef DeviceInfo;
Status = Device_QueryDeviceInfo(&Device, &DeviceInfo);
SWP_Profile_TypeDef SWP_ProfileIn, SWP_ProfileOut;
SWP_TraceInfo_TypeDef TraceInfo;
SWP_ProfileDeInit(&Device, &SWP_ProfileIn);
SWP_ProfileIn.StartFreq_Hz = 20000000;
SWP_ProfileIn.StopFreq_Hz = 300000000;
SWP_ProfileIn.RBW_Hz = 25000;
SWP_ProfileIn.RBWMode = RBW_Manual;
Status = SWP_Configuration(&Device, &SWP_ProfileIn, &SWP_ProfileOut, &TraceInfo);
vector<double> Frequency_Full(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBmFull(TraceInfo.FullsweepTracePoints);
vector<double> Frequency(TraceInfo.FullsweepTracePoints);
vector<float> PowerSpec_dBm(TraceInfo.FullsweepTracePoints);
int HopIndex = 0;
int FrameIndex = 0;
MeasAuxInfo_TypeDef MeasAuxInfo;
void* DSP = NULL;
DSP_Open(&DSP);
Status = SWP_GetFullSweep(&Device, Frequency_Full.data(), PowerSpec_dBmFull.data(),
&MeasAuxInfo);
uint32_t InterceptPoints;
DSP_InterceptSpectrum(SWP_ProfileOut.StartFreq_Hz, SWP_ProfileOut.StopFreq_Hz,
Frequency_Full.data(), PowerSpec_dBmFull.data(), TraceInfo.FullsweepTracePoints,
Frequency.data(), PowerSpec_dBm.data(), &InterceptPoints);
Frequency.resize(InterceptPoints);
PowerSpec_dBm.resize(InterceptPoints);
Status = Device_Close(&Device);
```

26. Structure Variable

26.1 BootProfile_TypeDef

PhysicalInterface_TypeDef **PhysicalInterface**	Specifies the physical bus type used by the device, determining the interface method and communication type between the device and the host. Refer to the definition of the PhysicalInterface_TypeDef enumeration structure for details.
DevicePowerSupply_TypeDef **DevicePowerSupply**	Specifies the power supply mode of the device, used for initializing the device's power interface. Refer to the definition of the DevicePowerSupply_TypeDef enumeration structure for details.
IPVersion_TypeDef **ETH_IPVersion**	Specifies the IP protocol version for the ETH interface, used for initializing network communication. Refer to the definition of the IPVersion_TypeDef enumeration structure for details.
uint8_t **ETH_IPAddress[16]**	IP address of the ETH interface, used for network communication configuration between the device and the host.
uint16_t **ETH_RemotePort**	Listening port number of the ETH interface, used for network connection and data transmission.
int32_t **ETH_ErrorCode**	Return code of the ETH interface, used to indicate network initialization or communication status.
int32_t **ETH_ReadTimeOut**	Read timeout for the ETH interface, measured in milliseconds (ms), used to control the waiting time for network data retrieval.

26.2 BootInfo_TypeDef

DeviceInfo_TypeDef **DeviceInfo**	Device basic information structure, containing the unique device identifier and various version information. Refer to the definition of the DeviceInfo_TypeDef structure for details.
uint32_t **BusSpeed**	Bus speed information, used to represent the rate of the device communication bus.
uint32_t **BusVersion**	Bus firmware version number.
uint32_t **APIVersion**	API version number.
int **ErrorCodes[7]**	List of error codes during the boot process, where each element represents an error type.
int **Errors**	Total number of errors during the boot process.
int **WarningCodes[7]**	List of warning codes during the boot process, where each element represents a warning type.
int **Warnings**	Total number of warnings during the boot process.

26.3 DeviceInfo_TypeDef

uint64_t **DeviceUID**	Unique device serial number, used to uniquely identify the device.
uint16_t **Model**	Device type or model identifier.
uint16_t **HardwareVersion**	Device hardware version number.
uint16_t **MFWVersion**	Device MCU firmware version number.
uint16_t **FFWVersion**	Device FPGA firmware version number.
uint16_t **PMUVersion**	Device PMU firmware version number.
uint16_t **AGUVersion**	Device AGU firmware version number.

26.4 PowerSupplyState_TypeDef

float **rf_vlotage**	RF board power port voltage (V).
float **rf_current**	RF board power port current (A).
float **usb_vlotage**	Digital board USB port voltage (V).
float **usb_current**	Digital board USB port current (A).

26.5 DeviceState_TypeDef

int16_t **Temperature**	Device temperature, measured in degrees Celsius; Actual Temperature = 0.01 × Temperature.
double **AbsoluteTimeStamp**	Absolute timestamp corresponding to the current data packet.
float **Latitude**	Latitude coordinates corresponding to the current data packet; positive for North, negative for South.
float **Longitude**	Longitude coordinates corresponding to the current data packet; positive for East, negative for West.
uint64_t **nsSinceEpoch**	Nanosecond-level system timestamp corresponding to the current data packet.

26.6 NetworkDeviceInfo_TypeDef

uint64_t **DeviceUID**	Device serial number, used to uniquely identify the device.
uint16_t **Model**	Device type number.
uint16_t **HardwareVersion**	Hardware version number.
uint32_t **MFWVersion**	MCU firmware version number.
uint32_t **FFWVersion**	FPGA firmware version number.
uint8_t **IPAddress[4]**	Device IP address (IPv4).
uint8_t **SubnetMask[4]**	Device subnet mask (IPv4).

26.7 HardWareState_TypeDef

GNSSPeriphType_TypeDef **GNSSPeriphType**	GNSS peripheral type. Refer to the definition of the GNSSPeriphType_TypeDef enumeration structure for details.
GNSSType_TypeDef **GNSSType**	GNSS receiver type. Refer to the definition of the GNSSType_TypeDef enumeration structure for details.
OCXOType_TypeDef **OCXOType**	OCXO type on the GNSS module. Refer to the definition of the OCXOType_TypeDef enumeration structure for details.
uint8_t **InternalOCXO**	Indicates whether the device's internal reference clock is an OCXO
uint8_t **SignalSourceEn**	Indicates whether the device supports signal generator functionality.
uint8_t **ADC_VariableRateEn**	Indicates whether the device supports variable ADC sampling rates.
uint8_t **IM3_filter**	Supplementary IF (Intermediate Frequency) filter (IM3 enhancement). 0: Disabled, 1: Enabled.

26.8 GNSSInfo_TypeDef

float **latitude**	Returns the latitude coordinate of the current position.
float **longitude**	Returns the longitude coordinate of the current position.
int16_t **altitude**	Returns the altitude (elevation) of the current position.
uint8_t **SatsNum**	Returns the number of satellites used for the current positioning.
uint8_t **GNSS_LockState**	Returns the GNSS lock status. 0: Unlocked, 1: Locked.
uint8_t **DOCXO_LockState**	Returns the DOCXO lock status. 0: Unlocked, 1: Locked.
DOCXOWorkMode_TypeDef **DOCXO_WorkMode**	Returns the DOCXO operating status. Refer to the definition of the DOCXOWorkMode_TypeDef enumeration structure for details.
GNSSAntennaState_TypeDef **GNSSAntennaState**	Returns the antenna status. Refer to the definition of the GNSSAntennaState_TypeDef enumeration structure for details.
int16_t **hour**	Return the hour part of the GNSS time and date information.
int16_t **minute**	Return the minute part of the GNSS time and date information.
int16_t **second**	Return the second part of the GNSS time and date information.
int16_t **Year**	Return the year part of the GNSS time and date information.
int16_t **month**	Return the month part of the GNSS time and date information.
int16_t **day**	Return the day part of the GNSS time and date information.

26.9 GNSS_SatDate_TypeDef

uint8_t **SatsNum_All**	Returns the number of visible satellites within the current range.
uint8_t **SatsNum_Use**	Returns the number of satellites used for positioning.
GNSS_SNR_TypeDef **GNSS_SNR_UsePos**	Returns the SNR information of the satellites used for positioning, including the maximum, minimum, and average SNR. Refer to the definition of the GNSS_SNR_TypeDef structure for details.
GNSS_SNR_TypeDef **GNSS_SNR_NotUsePos**	Return the SNR information of satellites that are in view but not used for positioning, such as the maximum, minimum, and average SNR. Refer to the definition of the GNSS_SNR_TypeDef structure for details.

26.10 GNSS_SNR_TypeDef

uint8_t **Max_SatxC_No**	Maximum SNR in the current signal set.
uint8_t **Min_SatxC_No**	Minimum SNR in the current signal set.
uint8_t **Avg_SatxC_No**	Average SNR of the current signal set.

26.11 SWP_Profile_TypeDef

double **StartFreq_Hz**	Start frequency in Hz. Range: 9 kHz to (Device Maximum Frequency - 100 Hz).
double **StopFreq_Hz**	Stop frequency in Hz. Range: (9 kHz + 100 Hz) to Device Maximum Frequency.
double **CenterFreq_Hz**	Center frequency in Hz. Range: 9 kHz to (StopFreq_Hz - 50 Hz).
double **Span_Hz**	Frequency span in Hz. Range: ≥ 100 Hz.
double **RefLevel_dBm**	Reference level in dBm. Range: -50 dBm to 23 dBm.
double **RBW_Hz**	Resolution Bandwidth in Hz. Range: 0.1 Hz to 10 MHz.
double **VBW_Hz**	Video Bandwidth in Hz. Range: 0.1 Hz to 10 MHz.
double **SweepTime**	When the scan time mode is set to Manual, this parameter represents the absolute time; when set to *N, this parameter represents the scan time multiplier. Range: 0.1s to 9999s.
SWP_FreqAssignment_TypeDef **FreqAssignment**	Set the frequency specification method, choosing either StartStop or CenterSpan to define the frequency. The default is StartStop. Refer to the definition of the SWP_FreqAssignment_TypeDef structure for details.
Window_TypeDef **Window**	Specify the window function to be used for FFT analysis. Defaults to the Blackman-Nuttall window. Refer to the definition of the Window_TypeDef enumeration structure for details.

RBWMode_TypeDef **RBWMode**	Set the RBW update mode. The default is RBW_Auto. Refer to the definition of the RBWMode_TypeDef enumeration structure for details.
VBWMode_TypeDef **VBWMode**	Set the VBW update mode. The default is VBW_TenTimesRBW. Refer to the definition of the VBWMode_TypeDef enumeration structure for details.
SweepTimeMode_TypeDef **SweepTimeMode**	Set the scan time mode. The default is SWTMode_minSWT. Refer to the definition of the SweepTimeMode_TypeDef enumeration structure for details.
Detector_TypeDef **Detector**	Set the detector. The default is Detector_PosPeak. Refer to the definition of the Detector_TypeDef enumeration structure for details.
TraceFormat_TypeDef **TraceFormat**	Set the trace format. The default is TraceFormat_Standard. Refer to the definition of the TraceFormat_TypeDef enumeration structure for details.
TraceDetectMode_TypeDef **TraceDetectMode**	Set the trace detection mode (frequency axis). The default is TraceDetectMode_Auto. Refer to the definition of the TraceDetectMode_TypeDef enumeration structure for details.
TraceDetector_TypeDef **TraceDetector**	Set the trace detector type. The default is TraceDetector_AutoSample. To specify a detector manually, first set TraceDetectMode to TraceDetectMode_Manual. Refer to the definition of the TraceDetector_TypeDef enumeration structure for details.
uint32_t **TracePoints**	Set the desired number of trace points. The system will automatically adjust based on the configured scan range and RBW, and return the actual available trace point count that is closest to the requested value.
TracePointsStrategy_TypeDef **TracePointsStrategy**	Set the trace point configuration strategy. The default is SweepSpeedPreferred. Refer to the definition of the TracePointsStrategy_TypeDef enumeration structure for details.
TraceAlign_TypeDef **TraceAlign**	Set the trace alignment method. The default is NativeAlign. Refer to the definition of the TraceAlign_TypeDef enumeration structure for details.
FFTExecutionStrategy_TypeDef **FFTExecutionStrategy**	Set the FFT execution strategy. The default is Auto. Refer to the definition of the FFTExecutionStrategy_TypeDef enumeration structure for details.
RxPort_TypeDef **RxPort**	Set the signal input port. This is only supported on instruments with a maximum frequency of 8.5GHz or below. Refer to the definition of the RxPort_TypeDef enumeration structure for details.

SpurRejection_TypeDef SpurRejection	Set the spurious suppression. The default is Standard. Higher suppression levels result in slower scan speeds. Refer to the definition of the SpurRejection_TypeDef enumeration structure for details.
ReferenceClockSource_TypeDef ReferenceClockSource	Set the reference clock source to 10MHz. Refer to the definition of the ReferenceClockSource_TypeDef enumeration structure for details.
double ReferenceClockFrequency	Set the reference clock frequency in Hz (only 10MHz is supported), allowing manual correction of the system clock accuracy.
uint8_t EnableReferenceClockOut	Enable reference clock output—only devices with a maximum frequency of 9GHz or above support 10MHz reference clock output. 0: Disable; 1: Enable (output 10MHz reference clock).
SystemClockSource_TypeDef SystemClockSource	Set the system clock source. The default is the internal system clock. Refer to the definition of the SystemClockSource_TypeDef enumeration structure for details.
double ExternalSystemClockFrequency	External system clock frequency in Hz. When using an external system clock source, this parameter must be set to 10MHz.
SWP_TriggerSource_TypeDef TriggerSource	Set the receiver's sweep input trigger source. The default is internal trigger in free-run mode. Refer to the definition of the SWP_TriggerSource_TypeDef enumeration structure for details.
TriggerEdge_TypeDef TriggerEdge	Set the input trigger edge. The default is rising edge trigger. Refer to the definition of the TriggerEdge_TypeDef enumeration structure for details.
TriggerOutMode_TypeDef TriggerOutMode	Set the trigger output mode. The default is no trigger output. Refer to the definition of the TriggerOutMode_TypeDef enumeration structure for details.
TriggerOutPulsePolarity_TypeDef TriggerOutPulsePolarity	Set the trigger output pulse polarity. The default is positive pulse. Refer to the definition of the TriggerOutPulsePolarity_TypeDef enumeration structure for details.
uint32_t PowerBalance	Set the dynamic power control in SWP mode. The default is 0. Typical range: 0–5000. Increasing this value reduces power consumption but also decreases scan speed.
GainStrategy_TypeDef GainStrategy	Set the gain strategy. The default is LowNoise. Refer to the definition of the GainStrategy_TypeDef enumeration structure for details.
PreamplifierState_TypeDef Preamplifier	Set the preamplifier operation. The default is automatic enable. Refer to the definition of the PreamplifierState_TypeDef enumeration structure for details.

uint8_t **AnalogIFBWGrade**	Set the IF bandwidth mode. The default is the SAW filter. 0: SAW filter, 100MHz bandwidth, 20% filter skirt, provides good in-band flatness; 1: LC filter, 110MHz bandwidth, 10% filter skirt, provides better out-of-band suppression.
uint8_t **IFGainGrade**	Set the IF gain level. The default is level 2. Devices with maximum frequency ≤8.5GHz: range 0 to 11; Devices with maximum frequency ≥9GHz: range 0 to 3.
int8_t **Atten**	Set the attenuation in dB to define the spectrum analyzer channel attenuation. The default is -1 (auto). Range: -1 to 33, When this value is not -1 (auto), it takes precedence over RefLevel_dBm. In this case: Reference Level = Attenuation – 10.
SWP_TraceType_TypeDef **TraceType**	Set the output trace type. The default is normal trace output. Refer to the definition of the SWP_TraceType_TypeDef enumeration structure for details.
LOOptimization_TypeDef **LOOptimization**	Set the local oscillator optimization. The default is automatic mode. Refer to the definition of the LOOptimization_TypeDef enumeration structure for details.

26.12 SWP_TraceInfo_TypeDef

int **FullsweepTracePoints**	Return the number of points in the full trace under the current configuration.
int **PartialsweepTracePoints**	Return the number of trace points per frequency point under the current configuration, i.e., the number of points retrieved each time GetPart is called.
int **TotalHops**	Return the number of frequency points in the full trace under the current configuration, i.e., the number of GetPart calls required for a complete trace.
uint32_t **UserStartIndex**	The array index in the trace array corresponding to the user-specified StartFreq_Hz. That is, when HopIndex = 0, Freq[UserStartIndex] is the frequency point closest to SWPProfile.StartFreq_Hz.
uint32_t **UserStopIndex**	The array index in the trace array corresponding to the user-specified StopFreq_Hz. That is, when HopIndex = TotalHops - 1, Freq[UserStopIndex] is the frequency point closest to SWPProfile.StopFreq_Hz.
double **TraceBinBW_Hz**	Return the frequency interval between two points in the trace under the current configuration.
double **StartFreq_Hz**	Frequency of the first point in the trace.

double **AnalysisBW_Hz**	Analysis bandwidth corresponding to each frequency point.
int **TraceDetectRatio**	Trace detection ratio, range: [1, 2^32-1] Defines the mapping ratio between the original sampled points and the output trace points. The trace detector extracts one or two valid data points from the original spectrum sequence every TraceDetectRatio points to form the output trace.
int **DecimateFactor**	Decimation factor of the time-domain data.
float **FrameTimeMultiple**	Frame analysis time multiplier. The analysis time at a single frequency point = default analysis time (set by the system) × frame time multiplier. Increasing the frame time multiplier increases the device's minimum scan time, but not in a strictly linear manner.
double **FrameTime**	Frame sweep time: the signal duration used for a single-frame FFT analysis (unit: s).
double **EstimateMinSweepTime**	Minimum sweep time configurable under the current settings. Unit: s. The value is mainly affected by Span, RBW, VBW, frame scan time, and other factors.
DataFormat_TypeDef **DataFormat**	Time-domain data format. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
uint64_t **SamplePoints**	Time-domain data sample length.
uint32_t **GainParameter**	Gain-related parameters, where the 3rd byte indicates the preamplifier state: PreAmplifierState(bits 23 to 16) 0: Off; 1: On.
DSPPlatform_Typedef **DSPPlatform**	Return the DSP computation platform used under the current configuration. Refer to the definition of the DSPPlatform_TypeDef enumeration structure for details.

26.13 MeasAuxInfo_TypeDef

uint32_t **MaxIndex**	Index of the maximum power in the data packet.
float **MaxPower_dBm**	Maximum power value in the data packet.
int16_t **Temperature**	Device temperature, in Celsius: Temperature * 0.01.
double **SysTimeStamp**	System timestamp, in seconds. After device power-on, the internal counter counts with an 8ns interval.
double **AbsoluteTimeStamp**	Absolute timestamp, provided by the system GNSS.

float **Latitude**	Latitude coordinate: positive for north, negative for south.
float **Longitude**	Longitude coordinate: positive for east, negative for west.
float **Altitude**	Altitude corresponding to the current data packet, in meters.
float **SATHealth**	Health (SNR) of the currently used GNSS satellites.
double **IFAGCGain**	Current IFAGC gain; positive values indicate amplification, negative values indicate attenuation, in dB.
double **RefClkFreqOffset**	Offset of the current device reference clock frequency, relative to the GNSS frequency, in ppm.
uint64_t **nsSinceEpoch**	Nanosecond-level system timestamp corresponding to the current data packet.

26.14 SWPTrace_TypeDef

double* **Freq_Hz**	Pointer to the start of the frequency array.
float* **PowerSpec_dBm**	Pointer to the start of the power array.
int **HopIndex**	Hopping frequency point index for stitching the spectrum.
int **FrameIndex**	Frame index of the data, used for applications such as quasi-peak detection.
void* **AlternIQStream**	Address of the time-domain data (interleaved IQ format). Raw IQ data without units.
float **ScaletoV**	Coefficient to convert time-domain data to absolute voltage (V). Raw IQ data * ScaletoV gives the IQ data in volts.
MeasAuxInfo_TypeDef **MeasAuxInfo**	Refer to the definition of the MeasAuxInfo_TypeDef structure for details.
SWP_Profile_TypeDef **SWP_Profile**	Refer to the definition of the SWP_Profile_TypeDef structure for details.
SWP_TraceInfo_TypeDef **SWP_TraceInfo**	Refer to the definition of the SWP_TraceInfo_TypeDef structure for details.
DeviceInfo_TypeDef **DeviceInfo**	Refer to the definition of the DeviceInfo_TypeDef structure for details.
DeviceState_TypeDef **DeviceState**	Refer to the definition of the DeviceState_TypeDef structure for details.

26.15 PNM_Profile_TypeDef

double **CenterFreq**	Set the fundamental frequency center, in Hz.
float **Threshold**	Set the carrier detection threshold, in dBm; carriers above this threshold are recognized.
double **RBWRatio**	RBW ratio (each segment RBW / segment start frequency), range: 0.01 to 0.3.

double StartOffsetFreq	Start frequency offset, range: 1Hz to 9MHz.
double StopOffsetFreq	Stop frequency offset, range: 10Hz to 10MHz.
uint32_t TraceAverage	Number of trace smoothing iterations.

26.16 PNM_MeasInfo_TypeDef

uint32_t Segments	Number of frequency segments (decade-based segmentation).
uint32_t TracePoints	Total number of trace points in the phase noise measurement results.
uint32_t PartialUpdateCounts	Number of trace refreshes during a single-phase noise analysis, i.e., the number of Get interface calls.
uint32_t FramesInSegment [PHASENOISE_MAXFREQSEGMENT]	Total frames for each segment. PHASENOISE_MAXFREQSEGMENT defines the maximum number of frequency segments (decade segments), with an upper limit of 7.
uint32_t FrameDetRatioOfSegment [PHASENOISE_MAXFREQSEGMENT]	Frame detection ratio corresponding to each segment
double StartFreqOfSegment [PHASENOISE_MAXFREQSEGMENT]	Start frequency corresponding to each segment.
double StopFreqOfSegment [PHASENOISE_MAXFREQSEGMENT]	Stop frequency corresponding to each segment.
double RBWOfSegment [PHASENOISE_MAXFREQSEGMENT]	RBW corresponding to each segment, in Hz.

26.17 IQS_Profile_TypeDef

double CenterFreq_Hz	Center frequency, range: 9kHz–(device maximum frequency to 50MHz).
double RefLevel_dBm	Reference level, range: -50dBm to +23dBm.
uint32_t DecimateFactor	Decimation factor of the time-domain data, range: 1 to 4096.
RxPort_TypeDef RxPort	Set the signal input port. Only applicable to devices with a maximum frequency of 8.5GHz or below that are equipped with the optional internal signal source. Refer to the definition of the RxPort_TypeDef enumeration structure for details.
uint32_t BusTimeout_ms	Set the data transfer timeout (ms), range: 1ms to 10s. If the IQ stream acquisition function does not receive data within the specified time, it is considered a timeout and returns an error.
IQS_TriggerSource_TypeDef TriggerSource	Set the input trigger source. The default is bus trigger. Refer to the definition of the IQS_TriggerSource_TypeDef enumeration structure for details.

TriggerEdge_TypeDef **TriggerEdge**	Set the input trigger edge. The default is rising edge. Refer to the definition of the TriggerEdge_TypeDef enumeration structure for details.
TriggerMode_TypeDef **TriggerMode**	Set the input trigger mode. The default is FixedPoints. Refer to the definition of the TriggerMode_TypeDef enumeration structure for details.
uint64_t **TriggerLength**	Set the trigger length, i.e., the number of data points acquired per trigger. Effective only when TriggerMode = FixedPoints Range: 32 to (UINT64_MAX/6) When using bus trigger with 16-bit IQ data, if TriggerLength does not exceed 33,263,616 points, data can be acquired and processed between two triggers.
TriggerOutMode_TypeDef **TriggerOutMode**	Set the trigger output mode. The default is rising edge. Refer to the definition of the TriggerOutMode_TypeDef enumeration structure for details.
TriggerOutPulsePolarity_TypeDef **TriggerOutPulsePolarity**	Set the trigger output pulse polarity. The default is rising edge. Refer to the definition of the TriggerOutPulsePolarity_TypeDef enumeration structure for details.
double **TriggerLevel_dBm**	Set the level trigger threshold. Effective only when TriggerSource = Level. Range: -70dBm to Reference Level..
double **TriggerLevel_SafeTime**	Level trigger debounces safety time, in seconds. Effective only when TriggerSource = Level. Range: 0 to (2^32 - 1) × (1.0/125 MHz) s. If the valid level of the trigger signal lasts less than this value, the trigger is ignored.
double **TriggerDelay**	Set the trigger delay, in seconds. After a trigger occurs, the action will be executed after this delay. Range: 0 to (2^32 – 1) × (1/125MHz) s.
double **PreTriggerTime**	Set the pre-trigger time, in seconds. Range: 0 to (2^16 – 1) × (1/125MHz)s. When a trigger occurs, data within this duration prior to the trigger is saved synchronously.
TriggerTimerSync_TypeDef **TriggerTimerSync**	Set the synchronization of the trigger timer. The default is synchronized with the rising edge of the external trigger. Refer to the definition of the TriggerTimerSync_TypeDef enumeration structure for details.
double **TriggerTimer_Period**	Set the timer trigger period, in seconds. Effective only when TriggerSource = Timer. Range: 0 to (2^32 – 1) × (1/125MHz)s.
uint8_t **EnableReTrigger**	Automatic retriggering: enables the device to perform additional timed triggers after the initial trigger source is activated. For example, after each external trigger, the device can automatically trigger 3 more times at 1ms intervals.

	Effective only when TriggerMode = FixedPoints: 0 = disabled, 1 = enabled.
double ReTrigger_Period	Automatic retrigger period: sets the interval for additional triggers after each main trigger, in seconds. Range: 0 to (2^32 – 1) × (1/125MHz)s.
uint16_t ReTrigger_Count	Automatic retrigger count: sets the number of additional triggers to execute after each main trigger. Range: 0 to (2^16 – 1) times.
DataFormat_TypeDef DataFormat	Set the output data format of IQ data. The default is 16-bit format. Refer to the definition of the DataFormat_TypeDef enumeration structure for details.
GainStrategy_TypeDef GainStrategy	Set the gain strategy. The default is LowNoise. Refer to the definition of the GainStrategy_TypeDef enumeration structure for details.
PreamplifierState_TypeDef Preamplifier	Set the preamplifier operation. The default is automatic enable. Refer to the definition of the PreamplifierState_TypeDef enumeration structure for details.
uint8_t AnalogIFBWGrade	Set the analog IF bandwidth mode. 0: SAW filter, 100MHz bandwidth, 20% filter skirt, provides good in-band flatness; 1: LC filter, 110MHz bandwidth, 10% filter skirt, provides better out-of-band suppression.
uint8_t IFGainGrade	Set the IF gain level. Higher levels correspond to higher IF gain. For instruments with a maximum frequency ≤8.5GHz: range 0 to 11. For instruments with a maximum frequency of 20GHz or 40GHz: range 0 to 3.
uint8_t EnableDebugMode	Debug mode. Not recommended for general use in advanced applications. The default value is 0.
ReferenceClockSource_TypeDef ReferenceClockSource	Set the reference clock source. The default is the 10MHz internal clock. Refer to the definition of the ReferenceClockSource_TypeDef enumeration structure for details.
double ReferenceClockFrequency	Set the reference clock frequency, in Hz. Only 10MHz reference clock is supported.
uint8_t EnableReferenceClockOut	Enable reference clock output. Only supported on devices with a maximum frequency of 9GHz or above. 0: Do not output; 1: Output 10 MHz reference clock.
SystemClockSource_TypeDef SystemClockSource	Set the system clock source. Refer to the definition of the SystemClockSource_TypeDef enumeration structure for details.
double ExternalSystemClockFrequency	External system clock frequency, in Hz.

double NativeIQSampleRate_SPS	Set the native IQ sampling rate. The device can adjust its sampling rate using this parameter. If changing this parameter has no effect, the device does not support adjusting its sampling rate.
uint8_t EnableIFAGC	IF AGC control. 0: AGC off, use MGC mode; 1: AGC on. If changing this parameter has no effect, the device does not support this feature.
int8_t Atten	Set the attenuation in dB to define the spectrum analyzer channel attenuation. The default is –1 (auto). Range: -1 to +33. When this value is not –1 (auto), it takes precedence over the reference level (RefLevel_dBm). In this case, the reference level is automatically set to: Attenuation – 10dBm.
DCCancelerMode_TypeDef DCCancelerMode	Applicable to specific devices. Set the DC suppressor mode, with the default being the high-pass filter enabled. Enable this function to suppress DC components if a DC offset appears at the center frequency. If no DC component is present, this parameter does not need to be set. Refer to the definition of the DCCancelerMode_TypeDef enumeration structure for details.
QDCMode_TypeDef QDCMode	Applicable to specific devices. Set the IQ amplitude and phase correction mode. If enabling the QDC function does not change the amplitude and phase characteristics of the IQ data, the device does not need to enable the QDC function. Refer to the definition of the QDCMode_TypeDef enumeration structure for details.
float QDCIGain	Applicable to specific devices. Set the normalized linear gain of the I channel. A value of 1.0 indicates no gain. Setting range: 0.8 to 1.2. Effective when QDCMode = QDCManualMode.
float QDCQGain	Applicable to specific devices. Set the normalized linear gain of the Q channel. A value of 1.0 indicates no gain. Setting range: 0.8 to 1.2. Effective when QDCMode = QDCManualMode.
float QDCPhaseComp	Applicable to specific devices. Set the phase compensation coefficient. Setting range: -0.2 to +0.2. Effective when QDCMode = QDCManualMode.
int8_t DCCIOffset	Applicable to specific devices. Set the DC offset of the I channel in LSB. Effective when DCCancelerMode = DCCManualOffsetMode
int8_t DCCQOffset	Applicable to specific devices. Set the DC offset of the Q channel in LSB.

	Effective when DCCancelerMode = DCCManualOffsetMode.
LOOptimization_TypeDef **LOOptimization**	Set the local oscillator optimization. The default is automatic mode. Refer to the definition of the LOOptimization_TypeDef enumeration structure for details.

26.18 IQS_StreamInfo_TypeDef

double **Bandwidth**	The bandwidth of the receiver's physical channel or digital signal processing under the current configuration.
double **IQSampleRate**	The corresponding single-channel IQ sampling rate under the current configuration, in S/s (samples per second).
uint64_t **PacketCount**	The total number of data packets acquired per trigger under the current configuration. Effective when TriggerMode = Fixedpoints.
uint64_t **StreamSamples**	When TriggerMode = Fixedpoints: indicates the total number of samples per trigger under the current configuration; When TriggerMode = Adaptive: this value has no physical meaning; the return value is 0.
uint64_t **StreamDataSize**	When TriggerMode = Fixedpoints: indicates the total number of data bytes per trigger under the current configuration; When TriggerMode = Adaptive: this value has no physical meaning; the return value is 0.
uint32_t **PacketSamples**	The number of sampling points contained in the data packets obtained each time IQS_GetIQStream is called.
uint32_t **PacketDataSize**	The number of valid data bytes obtained each time IQS_GetIQStream is called.
uint32_t **GainParameter**	Gain-related parameters, where the 3rd byte indicates the state of the pre-amplifier. PreAmplifierState (bits 23–16): 0 = off, 1 = on.

26.19 IQS/DET/RTA_TriggerInfo_TypeDef

uint64_t **SysTimerCountOfFirstDataPoint**	The system time counter value corresponding to the first data point in the data packet. After the device powers on, the internal timer starts counting with an 8ns period. The returned system timestamp is the value of this counter at that moment.
uint16_t **InPacketTriggeredDataSize**	The number of valid trigger data bytes in the data packet.
uint16_t **InPacketTriggerEdges**	The number of edges contained in the data.
uint32_t **StartDataIndexOfTriggerEdges[25]**	The offset of each trigger edge relative to the first point of the data packet.

uint64_t **SysTimerCountOfEdges[25]**	The system timestamp corresponding to each trigger edge in the data packet.
int8_t **EdgeType[25]**	The polarity of each trigger edge in the data packet.

26.20 IQStream_TypeDef

void* **AlternIQStream**	Return the time-domain IQ data packet (data within the packet is stored in the order IQIQIQ…). Each complete data packet has a fixed size of 64,968 bytes. When the IQ data type is set to int8_t, the I and Q channels each contain 32,484 points, with 1 byte per point; When the IQ data type is set to int16_t, the I and Q channels each contain 16,242 points, with 2 bytes per point.; When the IQ data type is set to int32_t, the I and Q channels each contain 8,121 points, with 4 bytes per point.
float **IQS_ScaleToV**	The coefficient used to convert the raw time-domain data acquired by the ADC into absolute voltage (V).
float **MaxPower_dBm**	Outputs the maximum power value in the current data packet, in dBm.
uint32_t **MaxIndex**	The index of the maximum power in the data packet, i.e., the position of the sample point corresponding to this value.
IQS_Profile_TypeDef **IQS_Profile**	Configuration information of the data. Refer to the definition of the IQS_Profile_TypeDef structure for details.
IQS_StreamInfo_TypeDef **StreamInfo**	Format information of the data. Refer to the definition of the IQS_StreamInfo_TypeDef structure for details.
IQS_TriggerInfo_TypeDef **TriggerInfo**	Trigger information of the data. Refer to the definition of the IQS/DET/RTA_TriggerInfo_TypeDef structure for details.
DeviceInfo_TypeDef **DeviceInfo**	Device information of the data. Refer to the definition of the DeviceInfo_TypeDef structure for details.
DeviceState_TypeDef **DeviceState**	Device status information of the data. Refer to the definition of the DeviceState_TypeDef structure for details.

26.21 DET_Profile_TypeDef

double CenterFreq_Hz	Refer to the definition of the IQS_Profile_TypeDef structure for details.
double RefLevel_dBm	
uint32_t DecimateFactor	
RxPort_TypeDef RxPort	
uint32_t BusTimeout_ms	
DET_TriggerSource_TypeDef TriggerSource	
TriggerEdge_TypeDef TriggerEdge	
TriggerMode_TypeDef TriggerMode	
uint64_t TriggerLength	
TriggerOutMode_TypeDef TriggerOutMode	
TriggerOutPulsePolarity_TypeDef TriggerOutPulsePolarity	
double TriggerLevel_dBm	
double TriggerLevel_SafeTime	
double TriggerDelay	
double PreTriggerTime	
TriggerTimerSync_TypeDef TriggerTimerSync	
double TriggerTimer_Period	
uint8_t EnableReTrigger	
double ReTrigger_Period	
uint16_t ReTrigger_Count	
Detector_TypeDef Detector	Set the detector. The default is sample detection. Refer to the definition of the Detector_TypeDef enumeration structure for details.
uint16_t DetectRatio	Set the DET trace detection ratio. The detector processes the power trace, generating one output trace point for every DetectRatio raw data points.
GainStrategy_TypeDef GainStrategy	Refer to the definition of the IQS_Profile_TypeDef structure for details.
PreamplifierState_TypeDef Preamplifier	

uint8_t AnalogIFBWGrade	
uint8_t IFGainGrade	
uint8_t EnableDebugMode	
ReferenceClockSource_TypeDef ReferenceClockSource	
double ReferenceClockFrequency	
uint8_t EnableReferenceClockOut	
SystemClockSource_TypeDef SystemClockSource	
double ExternalSystemClockFrequency	
int8_t Atten	
uint8_t EnableIFAGC	
DCCancelerMode_TypeDef DCCancelerMode	
QDCMode_TypeDef QDCMode	
float QDCIGain	
float QDCQGain	
float QDCPhaseComp	
int8_t DCCIOffset	
int8_t DCCQOffset	
LOOptimization_TypeDef LOOptimization	

26.22 DET_StreamInfo_TypeDef

uint64_t PacketCount	The total number of data packets acquired per trigger under the current configuration. Effective when TriggerMode = Fixedpoints.
uint64_t StreamSamples	When TriggerMode = Fixedpoints: indicates the total number of samples per trigger under the current configuration; When TriggerMode = Adaptive: this value has no physical meaning; the return value is 0.

uint64_t StreamDataSize	When TriggerMode = Fixedpoints: indicates the total number of data bytes per trigger under the current configuration;

	When TriggerMode = Adaptive: this value has no physical meaning; the return value is 0.
uint32_t PacketSamples	The number of data points contained in a single data packet, i.e., the number of samples in the data packet obtained each time DET_GetPowerStream is called.
uint32_t PacketDataSize	The number of data bytes contained in a single data packet, i.e., the number of bytes obtained each time DET_GetPowerStream is called.
double TimeResolution	Time resolution of the data points, in seconds.
uint32_t GainParameter	Gain-related parameters, where the 3rd byte indicates the state of the pre-amplifier. PreAmplifierState (bits 23–16): 0 = off, 1 = on.

26.23 ZSP_Profile_TypeDef

double CenterFreq_Hz	Refer to the definition of the IQS_Profile_TypeDef structure for details.
double RefLevel_dBm	
uint32_t DecimateFactor	
RxPort_TypeDef RxPort	
uint32_t BusTimeout_ms	
DET_TriggerSource_TypeDef TriggerSource	
TriggerEdge_TypeDef TriggerEdge	
TriggerMode_TypeDef TriggerMode	
uint64_t TriggerLength	
TriggerOutMode_TypeDef TriggerOutMode	
TriggerOutPulsePolarity_TypeDef TriggerOutPulsePolarity	
double TriggerLevel_dBm	
double TriggerLevel_SafeTime	
double TriggerDelay	
double PreTriggerTime	
TriggerTimerSync_TypeDef TriggerTimerSync	
double TriggerTimer_Period	
uint8_t EnableReTrigger	

double ReTrigger_Period	
uint16_t ReTrigger_Count	
Detector_TypeDef Detector	
uint16_t DetectRatio	
GainStrategy_TypeDef GainStrategy	
PreamplifierState_TypeDef Preamplifier	
uint8_t AnalogIFBWGrade	
uint8_t IFGainGrade	
uint8_t EnableDebugMode	
ReferenceClockSource_TypeDef ReferenceClockSource	
double ReferenceClockFrequency	
uint8_t EnableReferenceClockOut	
SystemClockSource_TypeDef SystemClockSource	
double ExternalSystemClockFrequency	
int8_t Atten	
DCCancelerMode_TypeDef DCCancelerMode	
QDCMode_TypeDef QDCMode	
float QDCIGain	
float QDCQGain	
float QDCPhaseComp	
int8_t DCCIOffset	
int8_t DCCQOffset	
LOOptimization_TypeDef LOOptimization	
double RBW_Hz	Resolution Bandwidth, in Hz.
double VBW_Hz	Video Bandwidth, in Hz.
VBWMode_TypeDef VBWMode	VBW update mode.

	Refer to the definition of the VBWMode_TypeDef enumeration structure for details.
RBWFilterType_TypeDef RBWFilterType	RBW filter type. Refer to the definition of the RBWFilterType_TypeDef enumeration structure for details.

26.24 RTA_Profile_TypeDef

double CenterFreq_Hz	Refer to the definition of the SWP_Profile_TypeDef structure for details.
double RefLevel_dBm	
double RBW_Hz	
double VBW_Hz	
RBWMode_TypeDef RBWMode	
VBWMode_TypeDef VBWMode	
uint32_t DecimateFactor	Set the decimation factor. Range: 2^n (n = 1 to 12).
Window_TypeDef Window	Refer to the definition of the SWP_Profile_TypeDef structure for details.
SweepTimeMode_TypeDef SweepTimeMode	
double SweepTime	
Detector_TypeDef Detector	
TraceDetectMode_TypeDef TraceDetectMode	
TraceDetector_TypeDef TraceDetector	
uint32_t TraceDetectRatio	Trace detection ratio. Range: [1, 2^32 - 1]. Defines the mapping ratio between the original sampling points and the output trace points. The trace detector extracts one valid data point from the original spectrum sequence for every TraceDetectRatio data points, and uses it as part of the output trace.
RxPort_TypeDef RxPort	Refer to the definition of the IQS_Profile_TypeDef structure for details.
uint32_t BusTimeout_ms	
RTA_TriggerSource_TypeDef TriggerSource	
TriggerEdge_TypeDef TriggerEdge	
TriggerMode_TypeDef TriggerMode	
double TriggerAcqTime	Defines the data acquisition duration after a single trigger event, unit: s. Range: ≥ 256 ns.

	Valid only when TriggerMode = Fixedpoints.
TriggerOutMode_TypeDef TriggerOutMode	Refer to the definition of the IQS_Profile_TypeDef structure for details.
TriggerOutPulsePolarity_TypeDef TriggerOutPulsePolarity	
double TriggerLevel_dBm	
double TriggerLevel_SafeTime	
double TriggerDelay	
double PreTriggerTime	
TriggerTimerSync_TypeDef TriggerTimerSync	
double TriggerTimer_Period	
uint8_t EnableReTrigger	
double ReTrigger_Period	
uint16_t ReTrigger_Count	
GainStrategy_TypeDef GainStrategy	
PreamplifierState_TypeDef Preamplifier	
uint8_t AnalogIFBWGrade	
uint8_t IFGainGrade	
uint8_t EnableDebugMode	
ReferenceClockSource_TypeDef ReferenceClockSource	
double ReferenceClockFrequency	
uint8_t EnableReferenceClockOut	
SystemClockSource_TypeDef SystemClockSource	
double ExternalSystemClockFrequency	
int8_t Atten	
uint8_t EnableIFAGC	
DCCancelerMode_TypeDef DCCancelerMode	
QDCMode_TypeDef QDCMode	

float QDCIGain	
float QDCQGain	
float QDCPhaseComp	
int8_t DCCIOffset	
int8_t DCCQOffset	
LOOptimization_TypeDef LOOptimization	

26.25 RTA_FrameInfo_TypeDef

double StartFrequency_Hz	Start frequency of the spectrum, unit: Hz.
double StopFrequency_Hz	Stop frequency of the spectrum, unit: Hz.
double POI	Minimum signal duration for 100% probability of intercept, unit: s.
double TraceTimestampStep	Timestamp increment between traces within each data packet. (The overall packet timestamp is SysTimerCountOfFirstDataPoint in TriggerInfo.)
double TimeResolution	Sampling interval of each time-domain data point, also the timestamp resolution.
double PacketAcqTime	Acquisition time corresponding to each data packet, unit: s.
uint32_t PacketCount	Total number of data packets corresponding to the current configuration. Valid only when TriggerMode = Fixedpoints.
uint32_t PacketFrame	Number of valid frames in each data packet.
uint32_t FFTSize	Number of points used for FFT per frame.
uint32_t FrameWidth	Number of valid points after FFT frame truncation, corresponding to the display points of each trace in the data packet. This can be used as the X-axis point count (width) of the probability density map.
uint32_t FrameHeight	Spectrum amplitude range corresponding to each FFT frame. This can be used as the Y-axis point count (height) of the probability density map.
uint32_t PacketSamplePoints	Number of acquisition points per data packet.
uint32_t PacketValidPoints	Number of valid frequency-domain data points in each data packet: PacketFrame × FrameWidth.
uint32_t MaxDensityValue	Maximum accumulated value of a single pixel in the probability density bitmap.
uint32_t GainParameter	Gain-related parameters, where the third byte indicates the preamplifier state. PreAmplifierState (23 to 16 Bit) 0: Off; 1: On.

26.26 RTA_PlotInfo_TypeDef

float **ScaleTodBm**	Compression caused by conversion from linear power to logarithmic power. The absolute power of the trace equals SpectrumStream[] × ScaleTodBm + OffsetTodBm.
float **OffsetTodBm**	Offset used to convert relative power to absolute power. The absolute power axis range (Y-axis) of the bitmap equals FrameHeight × ScaleTodBm + OffsetTodBm.
uint64_t **SpectrumBitmapIndex**	Acquisition count of the probability density map. When multiple probability density maps need to be overlaid, this field can be used as the overlay index.

26.27 Demod_Profile_TypeDef

uint64_t **SamplePoints**	Number of sampling points. Range: [16384, 320000]. It is recommended to set to 2000 × SamplingRate / SymbolRate. For demodulating 128QAM and 256QAM signals, it is recommended to set to 4000 × SamplingRate / SymbolRate.
double **SampleRate**	Sampling rate, Hz.
double **SymbolRate**	Symbol rate, sym/s. Range: [SamplingRate / 64, SamplingRate / 4].
Demod_ModType_TypeDef **ModType**	Sets the modulation type of the signal to be demodulated. Supported types: 2ASK / 2FSK / 4FSK / GMSK / BPSK / QPSK / 8PSK / 16QAM / 32QAM / 64QAM / 128QAM / 256QAM.
Demod_FilterType_TypeDef **FilterType**	Set the filter type. Currently, only Root Raised Cosine (RRC) filter is supported. Refer to the definition of the Demod_FilterType_TypeDef enumeration structure for details.
double **FilterAlpha**	Filter roll-off factor. Range: [0.01, 0.99].

26.28 DemodInfo_TypeDef

double* **eDiagram**	Starting memory address of the eye diagram data.
uint32_t **eDiagram_Len**	Length of the eye diagram data.
double* **I_constellation**	Starting memory address of the I-path constellation data.
double* **Q_constellation**	Starting memory address of the Q-path constellation data.
uint32_t **constellation_Len**	Length of the constellation data.
int32_t* **bitStream**	Starting memory address of the bitstream data.
uint32_t **bitStream_Len**	Length of the bitstream data.
int32_t* **symbol**	Starting memory address of the symbol stream data.

uint32_t symbol_Len	Length of the symbol stream data.
double* EVM	Starting memory address of the EVM data (also used for FSK Error and ASK Error).
uint32_t EVM_Len	Length of the EVM data.
double EVM_RMS	Root Mean Square (RMS) EVM, %.
double EVM_MAX	Peak EVM, %.
double* PhaseError	Starting memory address of phase error data, unit: degrees.
uint32_t PhaseError_Len	Length of phase error data.
double PhaseError_RMS	Root Mean Square (RMS) phase error, unit: degrees.
double PhaseError_MAX	Peak phase error, unit: degrees.
double* MagError	Starting memory address of amplitude error data.
uint32_t MagError_Len	Length of amplitude error data.
double MagError_RMS	Root Mean Square (RMS) amplitude error, %.
double MagError_MAX	Peak amplitude error, %.
double FreqError	Frequency error, carrier frequency offset relative to the center frequency, also CarrFreqOffset, unit: Hz.
double IQ_Offset	IQ offset, unit: dB, valid only in PSK and QAM modes.
double SNR	Signal-to-Noise Ratio (SNR), unit: dB, valid only in PSK and QAM modes.
double GainImb	IQ gain imbalance, unit: dB, valid only in PSK and QAM modes.
double QuadError	IQ quadrature skew error, unit: degrees, valid only in PSK and QAM modes.
double FSK_Deviation	FSK frequency offset, unit: Hz.
double CarrPower	Carrier power, unit: dBm, valid only in ASK mode.
double ASK_Depth	ASK modulation depth, %.
double AM_Depth	AM modulation depth, %.
double FM_Deviation	FM frequency deviation, unit: Hz.
double* Phase	Starting memory address of PM demodulated data.
uint32_t Phase_Len	Length of PM demodulated data.
double* Freq	Starting memory address of FM demodulated data.
uint32_t Freq_Len	Length of FM demodulated data.
double* Amp	Starting memory address of AM demodulated data.
uint32_t Amp_Len	Length of AM demodulated data.
double* SSB	Starting memory address of SSB demodulated data, used for both upper and lower sidebands.
uint32_t SSB_Len	Length of SSB demodulated data.

26.29 Demod_SymbolMap_TypeDef

float I	X-axis coordinate, representing the real part of the symbol in the complex plane.
float Q	Y-axis coordinate, representing the imaginary part of the symbol in the complex plane.

26.30 Pulse_Profile_TypeDef

uint32_t ExpPulseNum	Desired number of pulses to acquire. Range: [1, 500000].
Unit_TypeDef unit	Unit of pulse data, dBm or V. Refer to the definition of the Unit_TypeDef enumeration structure for details.
float* Pulse	Starting memory address of pulse data. The data unit depends on unit. When using the Pulse_Detect function, it should point to DET data. When using the Pulse_Detect_PM1 function, it should point to IQS data, and the data unit must be V.
uint64_t PulseSize	Length of pulse data. Range: [10, 500000000].
double TimeResolution_s	Time resolution of pulse data, unit: s.
double DetThreshold	Pulse detection threshold, in the same unit as the data.

26.31 PulseInfo_TypeDef

uint32_t ActPulseNum	Actual number of pulses acquired. Based on this number, the detection result of each pulse can be obtained from the pointer variable.
PulseTDParam_TypeDef* PulseTDParam	Starting memory address of pulse detection time-domain parameters. The structure includes pulse width, period, duty cycle, etc., with all units in seconds. Refer to the definition of the PulseTDParam_TypeDef structure for details.
PulseAMPParam_TypeDef* PulseAMPParam	Starting memory address of pulse detection amplitude parameters. The structure includes peak level, reference level, peak-to-base ratio, etc. Refer to the definition of the PulseAMPParam_TypeDef structure for details.
PulseEstParam_TypeDef* PulseEstParam	Starting memory address of pulse detection plotting parameters. Refer to the definition of the PulseEstParam_TypeDef structure for details.
PulseStatsParam_TypeDef PulseStats	Pulse detection statistics parameter structure, including minimum, maximum, and average period, etc., with all units in seconds. Refer to the definition of the PulseStatsParam_TypeDef structure for details.

26.32 PulseInfoPM1_TypeDef

uint32_t **ActPulseNum**	Actual number of pulses acquired. The detection result of each pulse can be obtained from the pointer variable based on this count.
PulseTDParam_TypeDef* **PulseTDParam**	Starting memory address of pulse detection time-domain parameters. The structure includes pulse width, period, duty cycle, etc., with all values in seconds. Refer to the definition of the PulseTDParam_TypeDef structure for details.
PulseAMPParam_TypeDef* **PulseAMPParam**	Starting memory address of pulse detection amplitude parameters. The structure includes peak level, reference level, peak-to-base ratio, etc. Refer to the definition of the PulseAMPParam_TypeDef structure for details.
PulseEstParam_TypeDef* **PulseEstParam**	Starting memory address of pulse detection plotting parameters. Refer to the definition of the PulseEstParam_TypeDef structure for details.
PulseStatsParam_TypeDef **PulseStats**	Pulse detection statistics parameter structure, including minimum, maximum, and average period, etc., with all units in seconds. Refer to the definition of the PulseStatsParam_TypeDef structure for details.
uint8_t* **Mod**	Pulse modulation type: 0 = CW, 1 = LFM.
PulseFreqPhaseParam_TypeDef* **PulseFreqPhase**	Frequency and phase information of the pulse. Refer to the definition of the PulseFreqPhaseParam_TypeDef structure for details.

26.33 PulseTDParam_TypeDef

double **RiseTime**	Rise time, unit: s.
double **RiseEdge**	Rising edge.
double **FallTime**	Fall time, unit: s.
double **FallEdge**	Falling edge.
double **Width**	Pulse width, unit: s.
double **Period**	Period, unit: s.
float **DutyCycle**	Duty cycle, %.

26.34 PulseAMPParam_TypeDef

float **TopLevel_dBm**	Peak level, unit: dBm.
float **TopLevel_V**	Peak level, unit: V.
float **BaseLevel_dBm**	Reference level, unit: dBm.
float **BaseLevel_V**	Reference level, unit: V.

float TopToBaseRatio_dB	Peak-to-base ratio, unit: dB.
float TopToBaseDiff_V	Peak-to-base difference, unit: V.
float Droop_dB	Droop, unit: dB.
float Droop_V	Droop, unit: V.
float Overshoot_dB	Overshoot, unit: dB.
float Overshoot_V	Overshoot, unit: V.
float Ripple_dB	Ripple, unit: dB.
float Ripple_V	Ripple, unit: V.

26.35 PulseEstParam_TypeDef

double Level_10pct_Index[2]	Array index of the 10% level value, 0 for the rising edge index, 1 for the falling edge index.
double Level_50pct_Index[2]	Array index of the 50% level value, 0 for the rising edge index, 1 for the falling edge index.
double Level_90pct_Index[2]	Array index of the 90% level value, 0 for the rising edge index, 1 for the falling edge index.
double Level_95pct_Index[2]	Array index of the 95% level value, 0 for the rising edge index, 1 for the falling edge index.
double Width_25pct_Index	Array index of the 25% pulse width position.
double Width_75pct_Index	Array index of the 75% pulse width position.
uint64_t Start_Index	Starting index of the inferred signal and noise data.
uint64_t Size	Length of the inferred signal and noise data.
float* Noise_dBm	Starting memory address of the inferred noise data, unit: dBm.
float* Noise_V	Starting memory address of the inferred noise data, unit: V.
float* Signal_dBm	Starting memory address of the inferred signal data, unit: dBm.
float* Signal_V	Starting memory address of the inferred signal data, unit: V.

26.36 PulseStatsParam_TypeDef

double MinPRI	Minimum period, unit: s.
double MaxPRI	Maximum period, unit: s.
double MeanPRI	Average period, unit: s.
double MinPW	Minimum pulse width, unit: s.
double MaxPW	Maximum pulse width, unit: s.
double MeanPW	Average pulse width, unit: s.
float PRIDeviationPercent	Period deviation percentage, %.
float PWDeviationPercent	Pulse width deviation percentage, %.

26.37 PulseFreqPhaseParam_TypeDef

double FreqMean	Mean frequency, unit: Hz.
double FreqErrorRMS	Frequency error.
double PhaseMean	Mean phase.
double PhaseErrorRMS	Phase error.

26.38 MSCAN_Profile_TypeDef

double CenterFreq_Hz	Center frequency, in Hz.
double RefLevel_dBm	Reference level, in dBm.
double DwellTime	Dwell time for each frequency point.
uint32_t DecimateFactor	Decimation factor of the time-domain data.
uint32_t FFTSize	Number of FFT points.
uint32_t DetectCount	Number of detections.
Detector_TypeDef Detector	Detection type. Refer to the definition of the Detector_TypeDef enumeration structure for details.
IFAGC_TypedDef IFAGC	Enable IFAGC. Refer to the definition of the IFAGC_TypeDef enumeration structure for details.
XPPSTrigger_TypedDef XPPSTrigger	Enable XPPSTrigger. Refer to the definition of the XPPSTrigger_TypeDef enumeration structure for details.
IQPlayBack_TypedDef IQPlayBack	Enable IQPlayBack. Refer to the definition of the IQPlayBack_TypeDef enumeration structure for details.
Window_TypeDef Window	Windows type. Refer to the definition of the Window_TypeDef enumeration structure for details.

26.39 MSCAN_Info_Typedef

int32_t SpectrumFrames	Number of frames in the spectrum.
int32_t SpectrumPoints	Number of points per spectrum frame.
int32_t IQStreamPoints	Number of points in the IQ data.
double CenterFreq_Hz	Center frequency, in Hz.
double Span_Hz	Frequency span, in Hz.
double IQSampleRate	IQ sampling rate under the current configuration.

26.40 MSCAN_Data_Typedef

int64_t RepeatIndex	Current repetition index of the Element.
int32_t ElementIndex	Index of the current Element.
int Status	Return status of the element processing.
uint32_t SpectrumFrames	Actual number of spectrum frames.
uint32_t SpectrumPoints	Number of points per spectrum frame.
uint32_t IQStreamPoints	Actual number of IQ data points.
float ScaleTodBm	Compression caused by converting linear power to logarithmic power. Absolute power = SpectrumStream[] × ScaleTodBm + OffsetTodBm.
float OffsetTodBm	Offset used to convert relative power to absolute power.
float ScaleToV	Coefficient for converting **int16** IQ data to absolute voltage (V).
uint8_t* SpectrumStream	Spectrum data (user needs to allocate memory; required size = frames × points).
int16_t* IQStream	IQ data (user needs to allocate memory; required size = number of IQ data points).
double Temperature	Device temperature, °C = 0.01 × Temperature.
double SysTimeStamp	System timestamp corresponding to the current data packet, in seconds.
double AbsoluteTimeStamp	Absolute timestamp corresponding to the current data packet.
double Latitude	Latitude corresponding to the current data packet; positive for north latitude and negative for south latitude.
double Longitude	Longitude corresponding to the current data packet; positive for east longitude and negative for west longitude.
double Altitude	Altitude corresponding to the current data packet, in meters.
double SATHealth	Health (SNR) of the currently used GNSS satellites.
double RefClkFreqOffset	Offset of the current device reference clock frequency relative to the GNSS frequency, in ppm.
uint64_t nsSinceEpoch	Nanosecond-level system timestamp corresponding to the current data packet.

26.41 ExtMixer_TypeDef

uint8_t Enable	External mixer enable flag.
uint8_t EnableLOOut	0: Preferentially uses TG as the LO; 1: preferentially uses LOOUT as the LO. If the preferred option is unavailable, the other option is automatically selected. The output structure returns the option actually selected.

uint8_t Harmonic	Harmonic number.
uint8_t IsHighLO	High-side/low-side LO selection.
double StartFreq_Hz	Start frequency (Hz).
double StopFreq_Hz	Stop frequency (Hz).
double IFFreq_Hz	Intermediate frequency (Hz).
double LOPower_dBm	LO power (dBm).

26.42 AMDemodParam_TypeDef

float* DemodWaveform	Demodulated waveform array, unit: %, length specified by DemodWaveformSize.
float* AFSpectrum_ModDepth	Audio spectrum amplitude array, length = DemodWaveformSize / 2, linear scale, unit: %.
double* AFSpectrum_Freq	Frequency array corresponding to the audio spectrum, unit: Hz.
uint32_t DemodWaveformSize	Number of sampling points of the demodulated waveform.
float ModDepth	Average modulation depth, unit: %.
float ModDepthPeakPos	Positive peak modulation depth (Peak+), unit: %.
float ModDepthPeakNeg	Negative peak modulation depth average (Peak−), unit: %.
float ModDepthHalfPeak	Half-peak modulation depth average ((Peak+ − Peak−) / 2), unit: %.
float ModDepthRMS	RMS modulation depth average, unit: %.
float CarrierPower	Carrier power, unit: dBm.
double ModRate	Modulation rate, unit: Hz.
float SINAD	Signal-to-noise and distortion ratio (SINAD), unit: dB.
float RMSPower	RMS power, unit: dBm.
double FreqError	Frequency error, unit: Hz.
float SNR	Signal-to-noise ratio (SNR), unit: dB.
float DistTotalVrms	Total distortion ratio (RMS), unit: %.
float THD	Total harmonic distortion (THD), unit: %.
float PEP	Peak Envelope Power (PEP), unit: dBm.

26.43 FMDemodParam_TypeDef

float* DemodWaveform	Demodulated waveform array, unit: Hz, length specified by DemodWaveformSize.
float* AFSpectrum_Deviation	Audio spectrum amplitude array, length = DemodWaveformSize / 2, linear scale, unit: Hz.
double* AFSpectrum_Freq	Frequency array corresponding to the audio spectrum, unit: Hz.
uint32_t DemodWaveformSize	Number of sampling points of the demodulated waveform.
float Deviation	Frequency deviation (Deviation), unit: Hz.

float **DeviationPeakPos**	Positive peak frequency deviation (Peak+), unit: Hz.
float **DeviationPeakNeg**	Negative peak frequency deviation average (Peak−), unit: Hz.
float **DeviationHalfPeak**	Half-peak frequency deviation average ((Peak+ − Peak−) / 2), unit: Hz.
float **DeviationRMS**	RMS frequency deviation average, unit: Hz.
float **CarrierPower**	Carrier power, unit: dBm.
double **CarrierFreqErr**	Carrier frequency error, unit: Hz.
double **ModRate**	Modulation rate, unit: Hz.
float **SINAD**	Signal-to-noise and distortion ratio (SINAD), unit: dB.
float **SNR**	Signal-to-noise ratio, unit: dB.
float **DistTotalVrms**	Total distortion ratio, unit: %.
float **THD**	Total harmonic distortion, unit: %.

26.44 ASG_Profile_TypeDef

double **CenterFreq_Hz**	Center frequency in fixed-point mode (SIG_Fixed), unit: Hz. Input range: 1 MHz to 1 GHz, step: 1 Hz.
double **Level_dBm**	Output power in fixed-point mode (SIG_Fixed), unit: dBm. Input range: -50 to 0 dBm, step: 0.25 dB.
double **StartFreq_Hz**	Start frequency in frequency sweep mode (SIG_FreqSweep_×), unit: Hz. Input range: 1 MHz to 1 GHz, step: 1 Hz.
double **StopFreq_Hz**	Stop frequency in frequency sweep mode (SIG_FreqSweep_×), unit: Hz. Input range: 1 MHz to 1 GHz, step: 1 Hz.
double **StepFreq_Hz**	Frequency step in frequency sweep mode (SIG_FreqSweep_×), unit: Hz. Input range: 1 Hz to 1 GHz, step: 1 Hz.
double **StartLevel_dBm**	Start power in power sweep mode, unit: dBm.
double **StopLevel_dBm**	Stop power in power sweep mode, unit: dBm.
double **StepLevel_dBm**	Power step in power sweep mode, unit: dBm.
double **DwellTime_s**	Dwell time in frequency sweep or power sweep mode, unit: s. Effective when the trigger source is BUS. Input range: 0 to 1000000, step: 1.
double **ReferenceClockFrequency**	Reference clock frequency setting, effective for both internal and external references.
ReferenceClockSource_TypeDef **ReferenceClockSource**	Reference clock source selection, used to specify whether to use the internal or external reference clock.

	Refer to the definition of the ReferenceClockSource_TypeDef enumeration structure for details.
ASG_Port_TypeDef **Port**	Output port selection of the analog signal source. Refer to the definition of the ASG_Port_TypeDef enumeration structure for details.
ASG_Mode_TypeDef **Mode**	Signal source operating mode selection. Refer to the definition of the ASG_Mode_TypeDef enumeration structure for details.
ASG_TriggerSource_TypeDef **TriggerSource**	Signal source trigger input configuration. The default setting is free-run. Refer to the definition of the ASG_TriggerSource_TypeDef enumeration structure for details.

ASG_TriggerInMode_TypeDef **TriggerInMode**	Signal source trigger input mode configuration. The default setting is single-point trigger. Refer to the definition of the ASG_TriggerInMode_TypeDef enumeration structure for details.
ASG_TriggerOutMode_TypeDef **TriggerOutMode**	Signal source trigger output mode configuration. The default setting is no output. Refer to the definition of the ASG_TriggerOutMode_TypeDef enumeration structure for details.

26.45 ASG_Info_TypeDef

uint32_t **SweepPoints**	Number of sweep points.

26.46 emi_scan_profile

double **start**	Scan start frequency, in Hz.
double **stop**	Scan stop frequency, in Hz.
float **ref_level**	Reference level, with unit specified by unit.
double **rbw**	RBW, in Hz.
float **filter_dB**	RBW definition point, typically 6 dB for EMI/CISPR.
double **vbw**	Video Bandwidth (VBW), in Hz.
float **dwell_time**	Dwell time per frequency point, in seconds (s).
emi_unit **unit**	Amplitude unit.
uint8_t **detector**	Sweep Detector, can be bitwise combined by emi_detector.
emi_window **window**	FFT window type.

26.47 emi_scan_info

double **first_freq**	Frequency of the first point of the trace, in Hz.
double **last_freq**	Frequency of the last point of the trace, in Hz.
uint32_t **total_points**	Total number of points in the complete trace.
uint32_t **sections**	Number of sweep sections.
uint32_t **section_points**	Number of points in a standard section.
uint32_t **last_section_points**	Number of points in the last section.

26.48 spur

double **freq**	Frequency of the highest power point within the over-threshold segment, in Hz.
float **level**	Amplitude of the highest power point within the over-threshold segment, unit is consistent with the input trace.
double **bw**	Bandwidth of the continuous over-threshold segment, in Hz.
float **over_db**	Amplitude by which the highest power point exceeds the detection threshold, in dB.

26.49 emi_measure_profile

double **center**	Measurement center frequency, in Hz.
float **ref_level**	Reference level, unit specified by unit.
double **bw**	Measurement bandwidth, in Hz.
float **filter_dB**	RBW definition point, typically 6 dB for EMI/CISPR.
float **charge_time**	Quasi-Peak detector charge time, in s.
float **decay_time**	Quasi-peak detector discharge time, in s.
float **measure_time**	Measurement time, in s.
emi_unit **unit**	Amplitude unit.
int8_t **mode**	Measurement Mode: 0 - measure based on measure_time; -1 - adaptive measurement, default is 2s but not exceeding 5s.

26.50 TraceAnalysisResult_IP3_TypeDef

double **LowToneFreq**	Low-tone signal frequency, unit depends on the data source.
double **HighToneFreq**	High-tone signal frequency, unit depends on the data source.
double **LowIM3PFreq**	Low-frequency third-order intermodulation product frequency, unit depends on the data source.

double HighIM3PFreq	High-frequency third-order intermodulation product frequency, unit depends on the data source.
float LowTonePower_dBm	Low-tone signal power, unit: dBm.
float HighTonePower_dBm	High-tone signal power, unit: dBm.
float TonePowerDiff_dB	Difference between low-tone and high-tone power.
float LowIM3P_dBc	Amplitude of the low-frequency third-order intermodulation product relative to the strongest fundamental signal, unit: dBc.
float HighIM3P_dBc	Amplitude of the high-frequency third-order intermodulation product relative to the strongest fundamental signal, unit: dBc.
float IP3_dBm	Calculated third-order intercept point (IP3), unit: dBm.

26.51 TraceAnalysisResult_IP2_TypeDef

double LowToneFreq	Low-tone signal frequency, unit depends on the data source.
double HighToneFreq	High-tone signal frequency, unit depends on the data source.
double IM2PFreq	Second-order intermodulation product frequency, unit depends on the data source.
float LowTonePower_dBm	Low-tone signal power, unit: dBm.
float HighTonePower_dBm	High-tone signal power, unit: dBm.
float TonePowerDiff_dB	Difference between low-tone and high-tone power.
float IM2P_dBc	Amplitude of the second-order intermodulation product relative to the strongest fundamental signal, unit: dBc.
float IP2_dBm	Calculated second-order intercept point (IP2), unit: dBm.

26.52 DSP_ChannelPowerInfo_TypeDef

float ChannelPower_dBm	Total power within the channel, unit: dBm.
float PowerDensity	Channel power density, unit: dBm/Hz.
float ChannelPeakIndex	Trace index corresponding to the power peak within the channel.
double ChannelPeakFreq_Hz	Frequency corresponding to the power peak within the channel, unit: Hz.
float ChannelPeakPower_dBm	Power peak within the channel, unit: dBm.

26.53 TraceAnalysisResult_XdB_TypeDef

double XdBBandWidth_Hz	XdB bandwidth, unit: Hz.
double CenterFreq_Hz	Center frequency corresponding to the XdB bandwidth, unit: Hz.
double StartFreq_Hz	Start frequency of the XdB bandwidth, unit: Hz.
double StopFreq_Hz	Stop frequency of the XdB bandwidth, unit: Hz.
float StartPower_dBm	Power at the start frequency of the XdB bandwidth, unit: dBm.

float StopPower_dBm	Power at the stop frequency of the XdB bandwidth, unit: dBm.
uint32_t PeakIndex	Trace index of the peak within the XdB bandwidth range.
double PeakFreq_Hz	Frequency of the peak within the XdB bandwidth range, unit: Hz.
float PeakPower_dBm	Power of the peak within the XdB bandwidth range, unit: dBm.

26.54 TraceAnalysisResult_OBW_TypeDef

double OccupiedBandWidth	Occupied bandwidth, unit: Hz.
double CenterFreq_Hz	Center frequency of the occupied bandwidth, unit: Hz.
double StartFreq_Hz	Start frequency of the occupied bandwidth, unit: Hz.
double StopFreq_Hz	Stop frequency of the occupied bandwidth, unit: Hz.
float StartPower_dBm	Power at the start frequency of the occupied bandwidth, unit: dBm.
float StopPower_dBm	Power at the stop frequency of the occupied bandwidth, unit: dBm.
float StartRatio	Power proportion at the start frequency of the occupied bandwidth.
float StopRatio	Power proportion at the stop frequency of the occupied bandwidth.
uint32_t PeakIndex	Index of the peak within the occupied bandwidth.
double PeakFreq_Hz	Frequency of the peak within the occupied bandwidth, unit: Hz.
float PeakPower_dBm	Power of the peak within the occupied bandwidth, unit: dBm.

26.55 DSP_ACPRFreqInfo_TypeDef

double RBW	Resolution bandwidth used for analysis, unit: Hz.
double MainChCenterFreq_Hz	Center frequency of the main channel, unit: Hz.
double MainChBW_Hz	Bandwidth of the main channel, unit: Hz.
double AdjChSpace_Hz	Adjacent channel spacing, the difference between the main channel center frequency and the adjacent channel center frequency, unit: Hz.
uint32_t AdjChPair	Number of adjacent channel pairs: 1 indicates one adjacent channel on each side, 2 indicates two adjacent channels on each side.

26.56 TraceAnalysisResult_ACPR_TypeDef

float MainChPower_dBm	Total power of the main channel, unit: dBm.
uint32_t MainChPeakIndex	Trace index corresponding to the peak of the main channel.
double MainChPeakFreq_Hz	Peak frequency of the main channel, unit: Hz.
float MainChPeakPower_dBm	Peak power of the main channel, unit: dBm.
double L_AdjChCenterFreq_Hz	Center frequency of the left adjacent channel, unit: Hz.
double L_AdjChBW_Hz	Bandwidth of the left adjacent channel, unit: Hz.
float L_AdjChPower_dBm	Power of the left adjacent channel, unit: dBm.

float L_AdjChPowerRatio	Left adjacent channel power ratio (Left Adjacent Power / Main Channel Power).
float L_AdjChPowerDiff_dBc	Left adjacent channel power difference (Left Adjacent Power – Main Channel Power), unit: dBc.
float L_AdjChPeakIndex	Trace index corresponding to the peak of the left adjacent channel.
double L_AdjChPeakFreq_Hz	Peak frequency of the left adjacent channel, unit: Hz.
float L_AdjChPeakPower_dBm	Peak power of the left adjacent channel, unit: dBm.
double R_AdjChCenterFreq_Hz	Center frequency of the right adjacent channel, unit: Hz.
double R_AdjChBW_Hz	Bandwidth of the right adjacent channel, unit: Hz.
float R_AdjChPower_dBm	Power of the right adjacent channel, unit: dBm.
float R_AdjChPowerRatio	Right adjacent channel power ratio (Right Adjacent Power / Main Channel Power).
float R_AdjChPowerDiff_dBc	Right adjacent channel power difference (Right Adjacent Power – Main Channel Power), unit: dBc.
float R_AdjChPeakIndex	Trace index corresponding to the peak of the right adjacent channel.
double R_AdjChPeakFreq_Hz	Peak frequency of the right adjacent channel, unit: Hz.
float R_AdjChPeakPower_dBm	Peak power of the right adjacent channel, unit: dBm.

26.57 DSP_FFT_TypeDef

uint32_t **FFTSize**	Number of FFT points.
uint32_t **SamplePts**	Number of valid sampling points.
Window_TypeDef **Window**	Window function specified for FFT analysis. The default is FlatTop window. Refer to the definition of the Window_TypeDef enumeration structure for details.
TraceDetector_TypeDef **TraceDetector**	Set the trace detector type. The default is positive peak detection. Refer to the definition of the TraceDetector_TypeDef enumeration structure for details.
uint32_t **DetectionRatio**	Trace detection ratio.
float **Intercept**	Output spectrum capture ratio. For example, Intercept = 0.8 means 80% of the spectrum is output.
bool **Calibration**	Set whether to perform calibration: 0 = no, 1 = yes.

26.58 DSP_DDC_TypeDef

double **DDCOffsetFrequency**	Complex mixing frequency offset used in the Digital Down Conversion (DDC) process, used to shift the target signal from the original center frequency to baseband.
double **SampleRate**	Sampling rate of the DDC output data, used to determine the data rate for downconversion and subsequent processing.
float **DecimateFactor**	Decimation factor for resampling, used to reduce data rate and bandwidth. Range: 1 to 216.
uint64_t **SamplePoints**	Number of valid sampling points in the DDC output, determining the length of data generated in a single downconversion process.

26.59 DSP_AudioAnalysis_TypeDef

double **AudioVoltage**	Effective voltage of the audio signal, unit: V.
double **AudioFrequency**	Fundamental frequency of the audio signal, unit: Hz.
double **SINDA**	Signal-to-noise and distortion ratio (SINAD) of the audio signal, unit: dB.
double **THD**	Total harmonic distortion (THD) of the audio signal, unit: %.

26.60 Filter_TypeDef

int **n**	Set the number of filter taps, n > 0.
float **fc**	Set the cutoff frequency, 0 < fc / SamplingRate < 0.5.
float **As**	Set the stopband attenuation, As > 0, unit: dB.
float **mu**	Set the fractional sample offset, -0.5 < mu < 0.5.
uint32_t **SamplePts**	Set the number of sampling points, Samples > 0.

26.61 DeviceFirmwareVersion_TypeDef

uint32_t **FFWVersion**	Device FPGA firmware version number.
uint32_t **MFWVersion**	Device MCU firmware version number.
uint32_t **BusVersion**	Device Bus firmware version number.
uint16_t **PMUVersion**	Device PMU firmware version number.
uint16_t **AGUVersion**	Device AGU firmware version number.

27. Structure Enumeration Variable

27.1 PhysicalInterface_TypeDef

USB	Use USB as the physical interface, applicable to USB-type devices.
ETH	Use 100M/1000M Ethernet as the physical interface, applicable to LAN-type devices.

27.2 DevicePowerSupply_TypeDef

USBPortAndPowerPort	Powered simultaneously via the USB data port and the power port.

27.3 IPVersion_TypeDef

IPv4	Use IPv4 address.

27.4 SysPowerState_TypeDef

PowerON	All system work areas powered on.
RFPowerOFF	RF powered off, cannot wake up quickly.
RFStandby	RF standby, can wake up quickly.

27.5 SysPowerMode_TypeDef

SysPowerMode_Normal	MCU power-on mode.
SysPowerMode_Standby	MCU standby mode.
SysPowerMode_Stop	MCU power-off mode.

27.6 FanState_TypeDef

FanState_On	Forced always on.
FanState_Off	Forced always off.
FanState_Auto	Automatic mode: automatically turns on when the device temperature ≥ 50°C and turns off when it drops to ≤ 40°C.

27.7 ClkCalibrationSource_TypeDef

CalibrateByExternal	Calibrate the clock using an external trigger signal.
CalibrateByGNSS1PPS	Calibrate the clock using the GNSS 1PPS signal.

27.8 GNSSDataSource_TypeDef

GNSSDataSource_Internal	Internal GNSS data source, referring to the GNSS module built into the device at the factory.
GNSSDataSource_External	External GNSS data source, referring to the manufacturer-developed GNSS module that can be connected to instruments without a built-in GNSS module.

27.9 GNSSPeriphType_TypeDef

GNSS_None	No peripherals.
GNSS_For_EIO	EIO peripheral.
GNSS_For_NX	NX peripheral.
GNSS_For_PX	PX peripheral.
GNSS_For_PXZ	PXZ peripheral.
GNSS_For_TG	TG peripheral.

27.10 GNSSType_TypeDef

None_GPS	No GPS receiver.
GNSS_GPS	Standard GPS.
GNSS_GPS_Pro	High-performance GPS.

27.11 OCXOType_TypeDef

None_OCXO	No OCXO.
GNSS_OCXO	Standard OCXO.
GNSS_DOCXO	Tunable OCXO.

27.12 GNSSAntennaState_TypeDef

GNSS_AntennaExternal	External antenna.
GNSS_AntennaInternal	Internal antenna.

27.13 DOCXOWorkMode_TypeDef

DOCXO_LockMode	Acquisition mode.
DOCXO_HoldMode	Tracking mode.

27.14 SWP_FreqAssignment_TypeDef

StartStop	Specify the scan range using start frequency and stop frequency.
CenterSpan	Specify the scan range using center frequency and sweep width.

27.15 Window_TypeDef

FlatTop	Provides good amplitude accuracy.
Blackman_Nuttall	Provides good frequency resolution.
LowSideLobe	Provides strong spectrum leakage suppression capability.
Rect	Rectangular window, equivalent to no windowing; the finite-length signal is directly truncated.
Kaiser	Kaiser window, where the main lobe width and sidelobe suppression can be configured as required; suitable for high dynamic range measurements.
Gaussian_CISPR	Used as a Gaussian window for the CISPR standard; supported only in EMC mode.

27.16 RBWMode_TypeDef

RBW_Manual	Manually enter RBW.
RBW_Auto	Automatically update RBW with SPAN.
RBW_OneThousandthSpan	Force RBW = 0.001 × SPAN.
RBW_OnePercentSpan	Force RBW = 0.01 × SPAN.

27.17 VBWMode_TypeDef

VBW_Manual	Manually enter VBW.
VBW_EqualToRBW	Force VBW = RBW.
VBW_TenPercentRBW	Force VBW = 0.1 × RBW.
VBW_OnePercentRBW	Force VBW = 0.01 × RBW.
VBW_TenTimesRBW	Force VBW = 10 × RBW, completely bypassing the VBW filter.

27.18 SweepTimeMode_TypeDef

SWTMode_minSWT	Perform a sweep with the minimum sweep time.
SWTMode_minSWTx2	Perform a sweep with approximately 2× the minimum sweep time.
SWTMode_minSWTx4	Perform a sweep with approximately 4× the minimum sweep time.
SWTMode_minSWTx10	Perform a sweep with approximately 10× the minimum sweep time.
SWTMode_minSWTx20	Perform a sweep with approximately 20× the minimum sweep time.
SWTMode_minSWTx50	Perform a sweep with approximately 50× the minimum sweep time.
SWTMode_minSWTxN	Perform a sweep with approximately N× the minimum sweep time, where N equals SweepTimeMultiple.
SWTMode_Manual	Perform a sweep with approximately the specified sweep time, where the sweep time equals SweepTime.
SWTMode_minSMPxN	Perform a single-frequency-point sweep with approximately N× the minimum sample time, where N equals SampleTimeMultiple.

27.19 Detector_TypeDef

Detector_Sample	No frame detection between power spectra of each frequency point.
Detector_PosPeak	Perform frame detection between power spectra of each frequency point; output a single frame, taking MaxHold across frames.
Detector_Average	Perform frame detection between power spectra of each frequency point; output a single frame, taking the average across frames.
Detector_NegPeak	Perform frame detection between power spectra of each frequency point; output a single frame, taking MinHold across frames.
Detector_MaxPower	Before FFT, perform long-time sampling at each frequency point and select the frame with the maximum power for FFT, used to capture pulses or other transient signals (SWP mode only).

Detector_RawFrames	Perform multiple samplings at each frequency point, analyze with multiple FFTs, and output power spectra frame by frame (SWP mode only).
Detector_RMS	Perform frame detection between power spectra of each frequency point; output a single frame, taking RMS across frames.

27.20 TraceFormat_TypeDef

TraceFormat_Standard	Frequency mapped at equal intervals, suitable for conventional spectrum observation and fast sweeps.
TraceFormat_PrecisFrq	Frequency mapped accurately, suitable for measurements requiring strict accuracy of signal frequency readings.

27.21 TraceDetectMode_TypeDef

TraceDetectMode_Auto	Automatically select trace detection mode.
TraceDetectMode_Manual	Specify trace detection mode.

27.22 TraceDetector_TypeDef

TraceDetector_AutoSample	Automatic sampling detection.
TraceDetector_Sample	Sampling detection.
TraceDetector_PosPeak	Positive peak detection.
TraceDetector_NegPeak	Negative peak detection.
TraceDetector_RMS	RMS detection.
TraceDetector_Bypass	No detection.
TraceDetector_AutoPeak	Automatic peak detection.
TraceDetector_Normal	Normal detection.

27.23 TracePointsStrategy_TypeDef

SweepSpeedPreferred	Prioritize achieving the fastest sweep speed while staying as close as possible to the target trace point count.
PointsAccuracyPreferred	Prioritize keeping the actual trace point count close to the target trace point count.
BinSizeAssined	Prioritize generating the trace according to the specified trace point count. Under this mapping strategy, the returned trace point count equals the actual issued trace point count. The frequency spacing = (stop frequency − start frequency) / (trace points − 1). This strategy allows control of the spacing between points in the returned trace (TraceBinBW_Hz), but the sweep speed is slower.

27.24 TraceAlign_TypeDef

NativeAlign	Natural alignment: Return spectrum data slightly wider than the issued frequency range.
AlignToStart	Align to start frequency: Precisely align the start of the frequency range, ensuring the spectrum data begins from the start frequency.

27.25 FFTExecutionStrategy_TypeDef

Auto	Automatically select CPU or FPGA for FFT computation based on settings. For RBW < 40 kHz, processing is done on the CPU; for RBW ≥ 40 kHz, processing is done on the FPGA. Processing includes VBW, detection, spurious suppression, FFT, and trace detection.
Auto_CPUPreferred	Automatically select CPU or FPGA for FFT computation based on settings, with CPU preferred.
Auto_FPGAPreferred	Automatically select CPU or FPGA for FFT computation based on settings, with FPGA preferred.
CPUOnly_LowResOcc	Force FFT computation on the CPU with low resource usage; maximum FFT points = 256K.
CPUOnly_MediumResOcc	Force FFT computation on the CPU with medium resource usage; maximum FFT points = 1M.
CPUOnly_HighResOcc	Force FFT computation on the CPU with high resource usage; maximum FFT points = 4M.
FPGAOnly	Force FFT computation on the FPGA; sweep speed may decrease when RBW is small.

27.26 RxPort_TypeDef

ExternalPort	The receiver receives data from an external input port.
InternalPort	The receiver receives RF signals from an internal auxiliary signal source. Applicable only when an internal signal source module is installed.

27.27 SpurRejection_TypeDef

Bypass	No spurious suppression.
Standard	Standard spurious suppression.
Enhanced	Enhanced spurious suppression.

27.28 ReferenceClockSource_TypeDef

ReferenceClockSource_Internal	Internal reference clock (default 10 MHz).
ReferenceClockSource_External	External reference clock (default 10 MHz); automatically switches to internal reference if the external reference fails to lock.

ReferenceClockSource_Internal_Premium	Internal high-quality clock source; available when an optional high-quality GNSS module is installed.
ReferenceClockSource_External_Forced	Force use of external reference clock; will not switch to internal reference even if locking fails.
ReferenceClockSource_External_SysClock	External system clock: the external clock input is directly used as the system clock, bypassing the internal reference PLL.

27.29 SystemClockSource_TypeDef

SystemClockSource_Internal	Use internal system clock source.
SystemClockSource_External	Use external system clock source; the external system clock frequency must be set to 10 MHz.

27.30 SWP_TriggerSource_TypeDef

InternalFreeRun	Internal trigger, free run.
ExternalPerHop	External trigger, advance one frequency point per trigger.
ExternalPerSweep	External trigger, refresh one trace per trigger.
InternelXppsPerHop	Internal GNSS pulse-per-second trigger: each trigger advances to the next frequency point.
InternelXppsPerSweep	Internal GNSS pulse-per-second trigger: each trigger refreshes one trace.
InternelXppsPerProfile	Internal GNSS pulse-per-second trigger: each trigger applies a new configuration.

27.31 TriggerEdge_TypeDef

RisingEdge	Trigger on rising edge.
FallingEdge	Trigger on falling edge.

27.32 TriggerOutMode_TypeDef

None	No trigger output.
PerHop	Output on completion of each frame.
PerSweep	Output on completion of each sweep.
PerProfile	Output on each configuration switch.

27.33 TriggerOutPulsePolarity_TypeDef

Positive	Positive pulse.
Negative	Negative pulse.

27.34 GainStrategy_TypeDef

LowNoisePreferred	Emphasize low noise: after setting, the preamplifier is automatically enabled, and the reference level is lowered to around –30 dBm.
HighLinearityPreferred	Emphasize high linearity: after setting, the preamplifier is forcibly disabled.

27.35 PreamplifierState_TypeDef

AutoOn	Automatically enable the preamplifier; the reference level is lowered to around –30 dBm, preamplifier on.
ForcedOff	Force the preamplifier to remain off.
OnLowGain	Amplifier low-gain mode.
OnMediumGain	Amplifier medium-gain mode.
OnHighGain	Amplifier high-gain mode.

27.36 SWP_TraceType_TypeDef

ClearWrite	Output the normal trace.
MaxHold	Output the trace with MaxHold applied.
MinHold	Output the trace with MinHold applied.
ClearWriteWithIQ	Output both time-domain and frequency-domain data for the current frequency point.

27.37 LOOptimization_TypeDef

LOOpt_Auto	LO optimization, automatic.
LOOpt_Speed	LO optimization, high sweep speed.
LOOpt_Spur	LO optimization, low spurious.
LOOpt_PhaseNoise	LO optimization, low phase noise.

27.38 DSPPlatform_Typedef

CPU_DSP	Compute on CPU.
FPGA_DSP	Compute on FPGA.

27.39 SWPApplication_TypeDef

SWPNoiseMeas	Display average noise level measurement.
SWPChannelPowerMeas	Channel power measurement.
SWPOBWMeas	Occupied bandwidth measurement.
SWPACPRMeas	Adjacent channel power ratio (ACPR) measurement.
SWPIM3Meas	IP3/IM3 measurement.

27.40 RxPort_TypeDef

ExternalPort	The receiver receives data from an external input port.
InternalPort	The receiver receives RF signals from an internal auxiliary signal source.

27.41 IQS_TriggerSource_TypeDef

External	External trigger: Triggered by a physical signal connected to the device's external trigger input port.
Bus	Bus trigger: Triggered via a function (command).
Level	Level trigger: The device monitors the input signal against a set level threshold and automatically triggers when the input exceeds the threshold.
Timer	Timer trigger: Use the device's internal timer to trigger automatically at a set time interval.
TxSweep	Internal signal source sweep trigger: Trigger data acquisition from the device's internal signal source sweep (requires ASG option). When selected, the acquisition is triggered by the output trigger signal from the signal source sweep.
MultiDevSyncByExt	External trigger synchronization: Multiple devices perform synchronized triggering upon arrival of an external trigger signal.
MultiDevSyncByGNSS1PPS	GNSS module 1PPS synchronization: Multiple devices perform synchronized triggering upon arrival of the 1PPS signal from the attached GNSS module.
GNSS1PPS	System GNSS 1PPS trigger: Use the 1PPS signal provided by the system GNSS for triggering (requires GNSS module).

27.42 TriggerMode_TypeDef

FixedPoints	After a trigger, acquire a fixed-length data segment (TriggerLength). During acquisition, further triggers are ignored. Upon completion, the device returns to the waiting-for-trigger state.
Adaptive	After a trigger, continuously acquire data until a stop signal is received (external trigger stop edge or bus trigger stop command).

27.43 TriggerTimerSync_TypeDef

NoneSync	Timer trigger not synchronized with external trigger.
SyncToExt_RisingEdge	Timer trigger synchronized with the rising edge of external trigger.
SyncToExt_FallingEdge	Timer trigger synchronized with the falling edge of external trigger.

SyncToExt_SingleRisingEdge	Timer trigger single-shot synchronized with the rising edge of external trigger (requires issuing a command for single-shot synchronization).
SyncToExt_SingleFallingEdge	Timer trigger single-shot synchronized with the falling edge of external trigger (requires issuing a command for single-shot synchronization).
SyncToGNSS1PPS_RisingEdge	Timer trigger synchronized with the rising edge of GNSS 1PPS (requires GNSS module).
SyncToGNSS1PPS_FallingEdge	Timer trigger synchronized with the falling edge of GNSS 1PPS (requires GNSS module).
SyncToGNSS1PPS_SingleRisingEdge	Timer trigger single-shot synchronized with the rising edge of GNSS 1PPS (requires issuing a command for single-shot synchronization and a GNSS module).
SyncToGNSS1PPS_SingleFallingEdge	Timer trigger single-shot synchronized with the falling edge of GNSS 1PPS (requires issuing a command for single-shot synchronization and a GNSS module).

27.44 DataFormat_TypeDef

Complex16bit	IQ data in 16-bit format
Complex32bit	IQ data in 32-bit format
Complex8bit	IQ data in 8-bit format
Real16bit	Real, single-channel data, 16-bit.
Real32bit	Real, single-channel data, 32-bit.
Real8bit	Real, single-channel data, 8-bit.
Realfloat	Real, single-channel data, 32-bit float.

27.45 DCCancelerMode_TypeDef

DCCOff	Disable DC suppression.
DCCHighPassFilterMode	Enable high-pass filter mode (provides better DC suppression but attenuates signal components from DC up to 100 kHz).
DCCManualOffsetMode	Enable manual bias mode: manually configure DCCIOffset and DCCQOffset bias values; suppression is weaker than high-pass filter mode but does not attenuate DC signal components.
DCCAutoOffsetMode	Enable automatic bias mode: automatically configure DCCIOffset and DCCQOffset bias values.

27.46 QDCMode_TypeDef

QDCOff	Disable QDC function.
QDCManualMode	Enable and use manual mode: configure based on user-set QDCIGain, QDCQGain, and QDCPhaseComp.
QDCAutoMode	Enable and use automatic QDC mode: automatically configure QDCIGain, QDCQGain, and QDCPhaseComp to default values.

27.47 RBWFilterType_TypeDef

RBWFilter_80PercentABW	Maximum bandwidth mode: Provides usable bandwidth at 80% of the sampling rate.
RBWFilter_Gaussian_3dB	Standard Gaussian filter: Bandwidth defined at −3 dB power points. Excellent transient response (no overshoot), the most commonly used default filter in spectrum analysis.
RBWFilter_Gaussian_6dB	Gaussian filter (−6 dB amplitude points): Bandwidth defined at −6 dB amplitude points. Mainly used for EMC/EMI testing compliant with CISPR standards.
RBWFilter_Gaussian_Impulse	Gaussian pulse filter: Used to accurately reproduce the time-domain waveform of pulse signals.
RBWFilter_Gaussian_Noise	Filter calibrated for equivalent noise bandwidth (ENBW): Used for precise measurement of noise power spectral density.
RBWFilter_Flattop	Flat-top filter: Used to eliminate amplitude errors caused by slight frequency offsets.

27.48 LookBack_TypeDef

LookBack_Off	Disable LookBack function, do not upload raw IQ data.
LookBack_On	Enable LookBack function, upload raw IQ data.

27.49 IFAGC_TypeDef

IFAGC_Off	IF automatic gain control off.
IFAGC_On	IF automatic gain control on.

27.50 XPPSTrigger_TypeDef

XPPSTrigger_Off	Disable XPPSTrigger.
XPPSTrigger_On	Enable XPPSTrigger.

27.51 IQPlayBack_TypeDef

IQPlayBack_Off	Disable IQPlayBack.
IQPlayBack_On	Enable IQPlayBack.

27.52 ASG_Port_TypeDef

ASG_Port_External	External port.
ASG_Port_Internal	Internal port.

27.53 ASG_Mode_TypeDef

ASG_Mute	Mute.
ASG_FixedPoint	Fixed points.
ASG_FrequencySweep	Frequency sweep.
ASG_PowerSweep	Power sweep.
ASG_TrackGeneraotr	Tracking signal generator function.

27.54 ASG_TriggerSource_TypeDef

ASG_TriggerIn_FreeRun	Free run.
ASG_TriggerIn_External	External trigger.
ASG_TriggerIn_Bus	Timer trigger.

27.55 ASG_TriggerInMode_TypeDef

ASG_TriggerInMode_Null	Free run.
ASG_TriggerInMode_SinglePoint	Single-point trigger (trigger once for a single frequency or power configuration).
ASG_TriggerInMode_SingleSweep	Single-sweep trigger (trigger once to perform one complete sweep cycle).
ASG_TriggerInMode_Continous	Continuous-sweep trigger (trigger once to operate continuously).

27.56 ASG_TriggerOutMode_TypeDef

ASG_TriggerOutMode_Null	Free run.
ASG_TriggerOutMode_SinglePoint	Single-point trigger (one pulse output per frequency hop).
ASG_TriggerOutMode_SingleSweep	Single-sweep trigger (one pulse output per sweep).

27.57 emi_detector

EMI_SAMPLE_DETECTOR	Sample detector.
EMI_MAX_DETECTOR	Peak detector.
EMI_RMS_DETECTOR	RMS detector.
EMI_MIN_DETECTOR	Min detector.
EMI_ALL_DETECTOR	Enable all detectors simultaneously. Multiple detectors can be combined using bitwise operations.

27.58 emi_window

CISPR	Use the FFT window function specified by the CISPR standard.

27.59 emi_unit

DBM	Amplitude unit: dBm.
DBMV	Amplitude unit: dBmV.
DBUV	Amplitude unit: dBμV.

27.60 search_mode

SPUR	Search for over-limit signals based on the limit line.
PEAK	Search for prominent signals based on the local noise floor.

27.61 Demod_FilterType_TypeDef

RootRaisedCosine	Root raised cosine filter.

27.62 Unit_TypeDef

Voltage_V	Voltage, V;
Power_dBm	Power, dBm.

Appendix 1: API Return Index

Code	Cause of error/warning	Type[1]	Handing
0	No error	-	No processing is required; subsequent processes can be executed normally.
-1	Bus open error	Error	Check the device power supply, data line connection and check if the driver is installed correctly. After troubleshooting the error, you need to call Device_Open again to open the device.
-3	RF calibration file is missing[2]	Error	Check if the RF calibration file is placed in the specified directory. After troubleshooting the error, you need to call Device_Open again to open the device.
-4	IF calibration file is missing[2]	Error	Check if the IF calibration file is placed in the specified directory. After eliminating the error, you need to call Device_Open again to open the device.
-5	Device configuration information is missing[2]	Error	Check if the RF calibration file used is correct with the IF calibration file. After removing the error, you need to call Device_Open again to open the device.
-6	Device specification file is missing[2]	Error	Check that the device specification file (if required) is placed in the specified directory.
-7	Update Strategy failed	Error	Re-call Device_Open to open the device.
-8	Bus communication error	Error	Re-call the configuration function in the current mode.
-9	Data content error	Error	Re-call the configuration function in the current mode.
-10	Data not retrieved within specified time	Warning	Check whether the trigger source outputs the trigger signal normally, if there is no abnormality, continue to call the current function until the data is obtained.
-11	Configuration error via bus down	Warning.	Re-call the Configuration function to configure the device.
-12	Input signal amplitude exceeds the rated range in the current configuration	Warning	The current function gets to reduce the input signal amplitude or increase RefLevel_dBm as appropriate.

-14	The temperature has changed significantly since the last configuration	Warning	The device temperature has changed significantly since the last configuration; it is recommended to re-call the Configuration function to configure the device for best performance.
-15	There is a locking exception in the local oscillator or clock	Warning	It is recommended to re-call the Configuration function to configure the device to try to restore the normal state.
-49	The current device firmware version and API version are not based on the same baseline version.	Warning	Refer to the baseline version correspondence table in Chapter 3 and update either the device firmware version or the API version accordingly.
10054	Device network connection disconnected	Error	Check the network configuration, ensure it is correct, and reconnect.
10060	Connection attempt failed; target host not responding	Error	Check the network configuration, ensure it is correct, and reconnect.
10062	Device did not acquire data correctly	Error	Check the network configuration, ensure it is correct, and reconnect.
-50	Pulse detection license file missing	Error	Place the xx_pulsedet.lic file in the /CalFile/ folder.
-51	Pulse detection license file content invalid	Error	Usually caused by license content being modified; please contact technical support.
-52	Pulse detection license file expired	Error	Please contact technical support.
-53	Incorrect number of pulses	Error	The expected number of pulses is 0; pulse detection will not be performed. Please adjust the number of pulses.
-54	Number of pulse detection data points below minimum limit	Error	Pulse detection will not be performed; please adjust the number of pulse detection data points.
50	Number of pulses exceeds limit	Warning	The expected number of pulses exceeds the upper limit; execution will follow the upper limit.
51	Number of pulse detection data points exceeds limit	Warning	Number of pulse detection data points exceeds the upper limit; execution will follow the upper limit.
-60	Demodulation license file missing	Error	Place the xx_demodlic.txt file in the /CalFile/ folder.

-61	Demodulation license file content invalid	Error	Usually caused by license content being modified; please contact technical support.
-62	Demodulation library missing	Error	Place DigitalSigDemod.dll (Windows) or libDigitalSigDemod.so (Linux) in the same path as the htra_api library.
-63	Failed to enable demodulation function	Error	This error usually does not occur; if it does, please contact technical support.
-64	Input sample points less than 16,384	Error	Increase the number of sample points to 16,384 or above.
-65	Input sample points greater than 320,000	Error	Reduce the number of sample points to 320,000 or below.
-66	Input sample rate/symbol rate less than 4	Error	Set the sample rate/symbol rate to 4 or higher.
-67	Input sample rate/symbol rate greater than 64	Error	Set the sample rate/symbol rate to 64 or lower.
-68	Input symbol rate less than or equal to 0	Error	Set the symbol rate to greater than 0.
-69	No demodulation data output	Error	When the input IQ data contains a large number of zeros, demodulation may fail.
-70	Demodulation failed	Error	Check whether the related parameter configuration is reasonable.
66	Sample points exceed limit	Warning	It is recommended that, while meeting the upper and lower limits, the number of sample points be set to ≥ 2000 × SampleRate / SymbolRate.
67	Filter coefficients exceed limit	Warning	Filter coefficients exceed the allowed range: [0.01, 0.99]; the default value 0.35 will be used.

[1]. Type is "Error", you need to troubleshoot the problem immediately and turn the device back on, otherwise the device cannot continue to run subsequent processes. If the type is "warning", the device can continue the process without shutting down or reopening the device. However, it is still recommended to deal with the specific return value and the current application scenario selectively.

[2]. For the return value of -3, -4, -5, or -6, you also need to confirm whether the file storage path is a full English path. If the path contains non-English characters, the API call will also indicate file loading failure.

Appendix 2: RTA PDF Bitmap Guide

Spectrum Trace Rendering

In RTA mode, users can render the spectrum trace using the SpectrumTrace[] array returned by the RTA_GetRealTimeSpectrum or RTA_GetRealTimeSpectrum_Raw function, together with the FrameInfo structure returned by the RTA_Configuration function.

1. Spectrum Data Structure Description

SpectrumStream[] is a concatenated collection of power data from multiple spectrum frames. Users must parse the array according to the parameters defined in the FrameInfo structure:

- Single-frame width (FrameInfo.FrameWidth): Number of power points contained in each trace;
- Total frame count (FrameInfo.PacketFrame): Number of traces included in the current return;
- Total valid points (FrameInfo.PacketValidPoints): Total length of the SpectrumStream[] array. The calculation formula is: FrameInfo.PacketValidPoints = FrameInfo.PacketFrame × FrameInfo.FrameWidth

2. Frequency Axis Rendering

In RTA mode, only the power-axis data is returned. The frequency axis must be generated by the user based on the returned start frequency, stop frequency, and the calculated frequency spacing. The rendering method is as follows:

- Start frequency: FrameInfo.StartFrequency_Hz;
- Stop frequency: FrameInfo.StopFrequency_Hz;
- Points per frame: FrameInfo.FrameWidth;

Frequency range:

FrequencyRange = FrameInfo.StopFrequency_Hz - FrameInfo.StartFrequency_Hz

Frequency step:

FrequencyStep = FrequencyRange / (FrameInfo.FrameWidth - 1)

3. Power Axis Rendering

The returned SpectrumStream[] array contains relative power values. It can be converted to absolute power as follows. There are FrameInfo.PacketFrame frames of data; only one frame needs to be extracted for rendering.

- Relative Power: SpectrumStream[];
- Absolute Power: SpectrumStream[] × PlotInfo.ScaleTodBm + PlotInfo.OffsetTodBm

When rendering the spectrum trace, you can display only the first frame in the UI by taking the first FrameInfo.FrameWidth points from the absolute power array.

Render probability density as a BitMap

The probability density map is obtained through the RTA_GetRealTimeSpectrum function using the SpectrumBitMap[] array. It is an array with a length of FrameInfo.FrameWidth × FrameInfo.FrameHeight. Each element of the probability density map array represents the hit count of the corresponding pixel in the probability density map.

1. Plotting Logic

Each FrameInfo.FrameWidth elements in SpectrumBitMap[] form a single row. The rows are rendered consecutively for FrameHeight rows in a left-to-right, top-to-bottom order;

2. Physical Rendering of Axes

- X-axis: Drawn according to the instructions in the Frequency Axis Rendering section.
- Y-axis: Rendered from bottom to top across FrameHeight rows. The y-value ranges from 0 to (FrameHeight - 1). The power value (unit: dBm) corresponding to each vertical pixel position y can be calculated as follows:

$$\mathrm{Power}_{\mathrm{dBm}} = \mathrm{y} * \mathrm{PlotInfo.\,ScaleTodBm} + \mathrm{PlotInfo.\,OffsetTodBm}$$

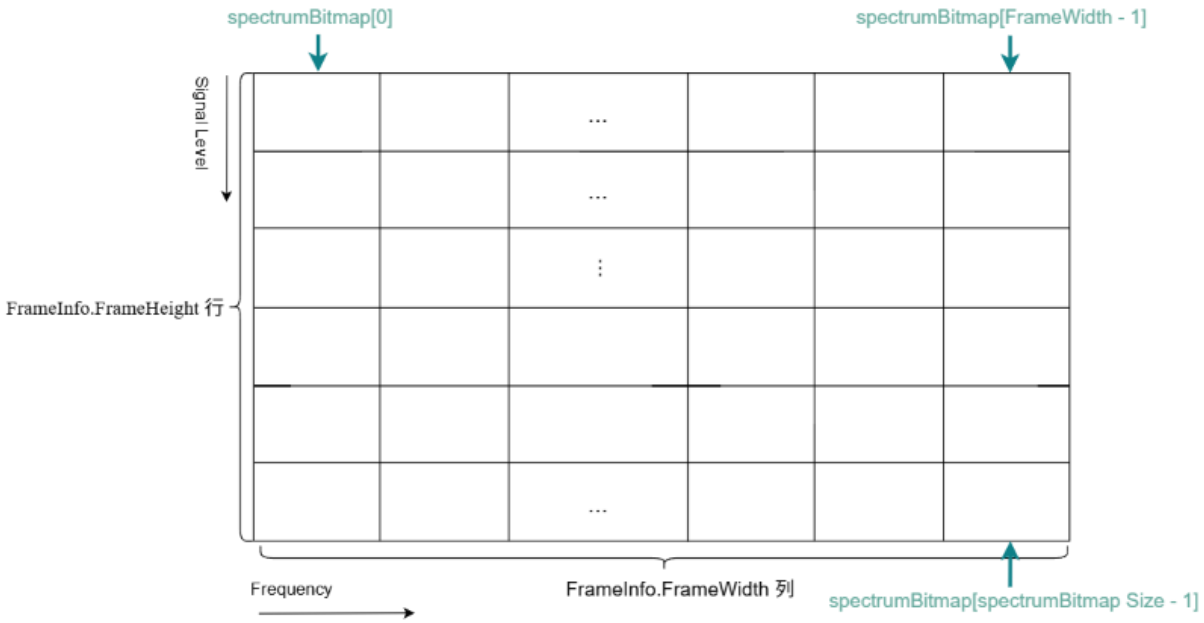


Figure 11 Probability Density Map Rendering Instructions

www.harogic.com
info@harogic.com
+65-8299 8857