Arbitrary Waveform Generator with FPGA Option

1 GS/s, 14 Bits, 2/4 Ch
500 MS/s, 16 Bits, 2/4 Ch
250 MS/s, 16 Bits, 4 Ch

Contact Signadyne for other form factors

CompactPCI Serial OpenVPX PCI Express µTCA ETHERNET USB

Features

- Options:
  - 1 GS/s, 14 bits, 2/4 ch, 400 MHz BW
  - 500 MS/s, 16 bits, 2/4 ch, 200 MHz BW
  - 250 MS/s, 16 bits, 4 ch, 100 MHz BW
- Embedded advanced Arbitrary Waveform Generators (AWGs):
  - Advanced triggering and marking (up to 8 reconfigurable I/O)
  - Waveform queue system with cycles, delays and prescalers
- Embedded high-precision Function Generators (FGs):
  - Sinusoidal, triangular, square, DC, etc.
  - 45-bit frequency resolution, 24-bit phase resolution
- Embedded ultra-flexible angle / amplitude modulators:
  - Modulations: AM, FM, PM, ASK, FSK, PSK, etc.
  - Simultaneous amplitude and angle modulations
- High-quality output signal with low phase noise:
  - SFDR: down to ∼ 70 dBc @ 120 MHz
  - Average Noise Density: down to ∼ -144 dBm/Hz
- Up to 2 GB of onboard RAM
- Hardware programming:
  - Signadyne HVI Technology (all models):
    - Ultra-fast real-time execution & decision making by HW
    - Built-in inter-module synchronization & data exchange
    - User-friendly flowchart-style programming
  - Onboard user-programmable FPGA (F models):
    - Xilinx Kintex-7 325T/410T FPGA
- Mechanical/Interface:
  - 1 slot 3U (PXIe)
  - Up to 1.6 GB/s transfer BW with P2P capabilities (PCIe Gen 2)
  - Independent DMA channels for fast and efficient data transfer

Programming Tools and Application Software

- Software programming:
  - Fully compatible programming libraries for most common languages, e.g. C, C++, C#, VB, LabVIEW, MATLAB, etc.
- Hardware programming:
  - HVI Programming:
    - Signadyne PROCESSflow, a graphical flowchart-style HVI programming environment
  - FPGA Programming (F models):
    - Graphical FPGA design with Signadyne FPGAflow
    - Seamless graphical programming with MATLAB/Simulink
    - VHDL and Verilog programming
- Signadyne VIRTUALknob: software front panels

Applications

- General purpose D/A, RF / arbitrary waveform generation
- Hardware-In-the-Loop (HIL) / ATE (Automated Test Equipment)
- R&D / Scientific equipment, aerospace & defense COTS

General Description

The SD AWG-H3300/H3300F Series are high-performance high-bandwidth AWGs with an advanced waveform generation system, flexible triggering, embedded function generators and modulators (frequency/phase/amplitude). Performance meets simplicity thanks to easy-to-use programming libraries, Signadyne HVI technology for real-time applications, and graphical FPGA programming (F models).

Product Table

Product	Analog Outputs				Features			FPGA Programming
	Speed (MSPS)	Bits	Ch	BW (MHz)	AM/FM/PM	IQ	ODSP[1]	
AWG-H3384(F)[3]	0.005-1000	14	4	DC-400	✓	-	-	✓[2]
AWG-H3383(F)[3]	0.005-1000	14	2	DC-400	✓	-	-	✓[2]
AWG-H3344	1000	14	4	DC-400	✓	-	-	-
AWG-H3343	1000	14	2	DC-400	✓	-	-	-
AWG-H3354(F)	0.005-500	16	4	DC-200	✓	-	-	✓[2]
AWG-H3353(F)	0.005-500	16	2	DC-200	✓	-	-	✓[2]
AWG-H3334	500	16	4	DC-200	✓	-	-	-
AWG-H3333	500	16	2	DC-200	✓	-	-	-
AWG-H3324(F)	0.005-250	16	4	DC-100	✓	-	-	✓[2]
Related Products								
AWG-H3500(F) Series	0.005-1000/500	14/16	4/2	DC-400/200	✓	✓	✓	✓[2]

[1] Onboard DSP (ODSP): DUCs, upsamplers, CIC/FIR filters [2] F model [3] Preliminary product, contact Signadyne for delivery time

Functional Block Diagram

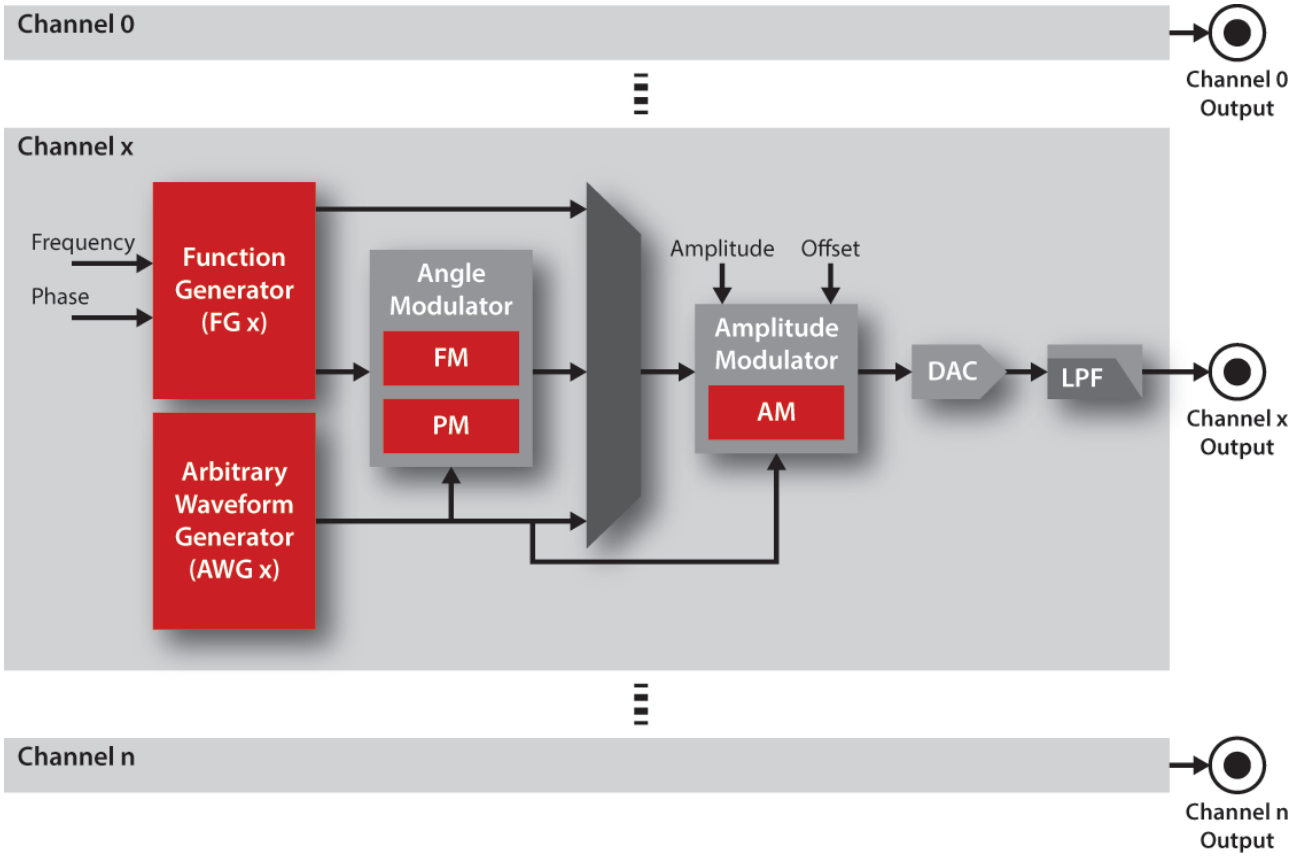


Figure 1: SD AWG-H3300/H3300F Series functional block diagram. All channels have identical output structure.

Ordering Information

Product Name	Description
SD AWG-H3384-XXXX-YYYY	1 GS/s, 14 Bits, 4 Ch, Arbitrary Waveform Generator
SD AWG-H3384F-XXXX-YYYY	1 GS/s, 14 Bits, 4 Ch, Arbitrary Waveform Generator with FPGA Programming
SD AWG-H3383-XXXX-YYYY	1 GS/s, 14 Bits, 2 Ch, Arbitrary Waveform Generator
SD AWG-H3383F-XXXX-YYYY	1 GS/s, 14 Bits, 2 Ch, Arbitrary Waveform Generator with FPGA Programming
SD AWG-H3344-XXXX-YYYY	1 GS/s, 14 Bits, 4 Ch, Arbitrary Waveform Generator
SD AWG-H3343-XXXX-YYYY	1 GS/s, 14 Bits, 2 Ch, Arbitrary Waveform Generator
SD AWG-H3354-XXXX-YYYY	500 MS/s, 16 Bits, 4 Ch, Arbitrary Waveform Generator
SD AWG-H3354F-XXXX-YYYY	500 MS/s, 16 Bits, 4 Ch, Arbitrary Waveform Generator with FPGA Programming
SD AWG-H3353-XXXX-YYYY	500 MS/s, 16 Bits, 2 Ch, Arbitrary Waveform Generator
SD AWG-H3353F-XXXX-YYYY	500 MS/s, 16 Bits, 2 Ch, Arbitrary Waveform Generator with FPGA Programming
SD AWG-H3334-XXXX-YYYY	500 MS/s, 16 Bits, 4 Ch, Arbitrary Waveform Generator
SD AWG-H3333-XXXX-YYYY	500 MS/s, 16 Bits, 2 Ch, Arbitrary Waveform Generator
SD AWG-H3324-XXXX-YYYY	250 MS/s, 16 Bits, 4 Ch, Arbitrary Waveform Generator
SD AWG-H3324F-XXXX-YYYY	250 MS/s, 16 Bits, 4 Ch, Arbitrary Waveform Generator with FPGA Programming

Options (XXXX)	Form Factor
PXIe	PXI Express (CompactPCI Express compatible)
cPCIs	CompactPCI Serial[1]
VPX	VPX (OpenVPX)[1]
PCIe	PCI Express[1]
MTCA	MTCA.4[1]
ETH	Ethernet[1]
USB	USB[1]

[1] Contact Signadyne for these and other form factors

Options (YYYY)	Onboard Memory (RAM)[1]
2G	2 GByte
256M	256 MBytes
128M	128 MBytes
32M	32 MBytes
16M	16 MBytes

[1] Contact Signadyne for other memory configurations

Specifications

Specifications Summary (SD AWG-H3350(F) / H3380(F))

	AWG-H3353(F)			AWG-H3354(F)			AWG-H3383(F)			AWG-H3384(F)				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
Inputs and Outputs														
Channels (SE mode)		2			4			2			4		Out	
Channels (Diff. mode)		1			2			2			4		Out	
Reference Clock[1]		1			1			1			1		Out	
Reference Clock[2]		1			1			1			1		In	
Triggers/Markers[1,3]		1			1			8			8		In/Out	Reconfig.
Triggers/Markers[2,3]		8			8			8			8		In/Out	Reconfig.
Output Channels Overview														
Sampling Rate	0.005		500	0.005		500	0.005		1000	0.005		1000	MSPS	
Voltage Resolution		16			16			14			14		Bits	
Output Frequency	DC		200	DC		200	DC		400	DC		400	MHz	
Real-time BW		200			200			400			400		MHz	
Output Voltage	-1.5		1.5	-1.5		1.5	-1		1	-1		1	Volts	
Built-in Functionalities														
Function Generators		2			4			2			4			1 per ch
Dual AWGs		2			4			2			4			1 per ch
Frequency Modulators		2			4			2			4			1 per ch
Phase Modulators		2			4			2			4			1 per ch
Amplitude Modulators		2			4			2			4			1 per ch
DC Offset Modulators		2			4			2			4			1 per ch
Onboard Memory														
RAM Memory	16		2048	32		2048	16		2048	32		2048	MBytes	

[1] At front panel
[2] At backplane
[3] Markers available from firmware version v3.0 or later

Table 1: General Overview (SD AWG-H3350(F) / H3380(F))

Specifications Summary (SD AWG-H3330 / H3340)

	AWG-H3333			AWG-H3334			AWG-H3343			AWG-H3344				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
Inputs and Outputs														
Channels (SE mode)		2			4			2			4		Out	
Channels (Diff. mode)		1			2			1			2		Out	Diff. uses 2ch
Reference Clock[1]		1			1			1			1		Out	
Reference Clock[2]		1			1			1			1		In	
Triggers/Markers[1,3]		1			1			1			1		In/Out	Reconfig.
Triggers/Markers[2,3]		8			8			8			8		In/Out	Reconfig.
Output Channels Overview														
Sampling Rate		500			500			1000			1000		MSPS	
Voltage Resolution		16			16			14			14		Bits	
Output Frequency	DC		200	DC		200	DC		400	DC		400	MHz	
Real-time BW		200			200			400			400		MHz	
Output Voltage	-1.5		1.5	-1.5		1.5	-1		1	-1		1	Volts	
Built-in Functionalities														
Function Generators		2			4			2			4			1 per ch
Dual AWGs		2			4			2			4			1 per ch
Frequency Modulators		2			4			2			4			1 per ch
Phase Modulators		2			4			2			4			1 per ch
Amplitude Modulators		2			4			2			4			1 per ch
DC Offset Modulators		2			4			2			4			1 per ch
Onboard Memory														
RAM Memory	16		1024	16		1024	16		1024	16		1024	MBytes	

[1] At front panel
[2] At backplane
[3] Markers available from firmware version v3.0 or later

Table 2: General Overview (SD AWG-H3330 / H3340)

Environmental Specifications (PXI Express)

	AWG-H3380(F) / H3350(F)			AWG-H3340 / H3330				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
System Bus								
Slots		1			1		Slot	PXI Express (CompactPCI Express compatible)
PCI Express Type			Gen 2			Gen 1	-	Automatic Gen negotiation, chassis dependent
PCI Express Link	1		4	1		4	Lanes	Automatic lane negotiation, chassis dependent
PCI Express Speed	400		1600	200		350	MBytes/s	Depends on # of lanes, chassis, congestion, etc.
Sustainable Throughput	200		800	100		170	MPoints/s	Depends on # of lanes, chassis, congestion, etc.
Power and Temperature								
3.3V PXIe Power Supply			1.5			1.5	A	$\sim$ 5 W
12V PXIe Power Supply			2			2	A	$\sim$ 24 W
Operating Temperature	-20		+55	-20		+55	^{o}C	In a standard chassis with forced ventilation

Table 3: Environmental Specifications (PXI Express)

I/O Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
Output Channels								
Sampling Rate		500			1000		MSPS	
Output Frequency	DC		200	DC		400	MHz	Limited by a reconstruction filter
Output Voltage	-1.5		1.5	-1		1	Volts	On a 50Ω load
Source Impedance		50			50		Ω	
Reference Clock Output								
Frequency		10 or 100			10 or 100		MHz	Generated from the internal clock. User selectable
Voltage		800			800		mV_{pp}	On a 50 Ω load
Power		2			2		dBm	On a 50 Ω load
Source Impedance		50			50		Ω	AC coupled
External I/O Trigger/Marker								
V_{IH}	2		5	2		5	V	
V_{IL}	0		0.8	0		0.8	V	
V_{OH}	2.4		3.3	2.4		3.3	V	On a high Z load
V_{OL}	0		0.5	0		0.5	V	On a high Z load
Input Impedance		10			10		KΩ	
Source Impedance		TTL			TTL		–	
Speed			500			500	Mbps	

Table 4: Input/Output Specifications

Function Generators (FGs) Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Function Generators		2 / 4			2 / 4		-	1 per channel
Waveform Types		4			4		-	Sinusoidal, triangular, square and DC
Frequency Range	0		200	0		400	MHz	
Frequency Resolution		45			45		Bits	
Frequency Resolution		5.7			11.4		μHz	
Phase Range	0		360	0		360	Deg	
Phase Resolution		24			24		Bits	
Phase Resolution		21.5			21.5		μdeg	
Speed Performance								
Frequency Change Rate			100			100	MChanges/s	With HVI Technology (Section 2.1.1 on page 30)
Frequency Modulation Rate			500			1000	MSamples/s	With AWGs and Angle Modulators
Phase Change Rate			100			100	MChanges/s	With HVI Technology (Section 2.1.1 on page 30)
Phase Modulation Rate			500			1000	MSamples/s	With AWGs and Angle Modulators

Table 5: Function Generators (FGs) Specifications

Amplitude and Offset Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Amplitude / Offset Range	-1.5		1.5	-1		1	Volts	Amplitude + Offset values
Amplitude / Offset Resolution		16			14		Bits	
Amplitude / Offset Resolution		45.8			122.1		μV	
Speed Performance								
Amplitude / Offset Change Rate			500			1000	MChanges/s	With HVI Tech. (Sect. 2.1.1 on page 30)
Amplitude / Offset Modulation Rate			500			1000	MSamples/s	With AWGs and Amplitude Modulators

Table 6: Amplitude and Offset Specifications

Arbitrary Waveform Generators (AWGs) Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Dual AWGs		2 / 4			2 / 4			1 Dual AWG per output channel
Aggregated Speed (16 bits)			4000			4000	MSPS	For all onboard waveforms combined
Aggregated Speed (32 bits)			2000			2000	MSPS	For all onboard waveforms combined
Waveform Multiple		5			10		Samples	Waveform length must be a multiple of this value
16-bit Waveform Length	15		478M	30		478M	Samples	Maximum depends on onboard RAM
32-bit Waveform Length	10		239M	20		239M	Samples	Maximum depends on onboard RAM
Waveform Length Efficiency		93.5			93.5		%	Effic. = Waveform Size / Waveform Size in RAM
Trigger		selec.			selec			External Trigger (input connector, backplane triggers), Software Trigger
AWG Specifications (16-bit Single Waveform)								
Speed			500			1000	MSPS	Per AWG
Resolution		16			16		Bits	
AWG Destination		Selec.			Selec.			Amplitude, Offset, Frequency or Phase
AWG Specifications (16-bit Dual Waveform)								
Speed (Waveform A)			500			1000	MSPS	Per AWG
Speed (Waveform B)			500			1000	MSPS	Per AWG
Resolution (Waveform A)		16			16		Bits	
Resolution (Waveform B)		16			16		Bits	
AWG Destination (Waveform A)		Selec.			Selec.			Amplitude or Offset
AWG Destination (Waveform B)		Selec.			Selec.			Frequency or Phase
AWG Specifications (32-bit Single Waveform)								
Speed			100			100	MSPS	Per AWG. Minimum prescaler: 1
Resolution		32			32		Bits	
AWG Destination		Selec.			Selec.			Amplitude, Offset, Frequency or Phase
AWG Specifications (32-bit Dual Waveform)								
Speed (Waveform A)			100			200	MSPS	Per AWG. Minimum prescaler: 1
Speed (Waveform B)			100			200	MSPS	Per AWG. Minimum prescaler: 1
Resolution (Waveform A)		32			32		Bits	
Resolution (Waveform B)		32			32		Bits	
AWG Destination (Waveform A)		Selec.			Selec.			Amplitude or Offset
AWG Destination (Waveform B)		Selec.			Selec.			Frequency or Phase

Table 7: Arbitrary Waveform Generators (AWGs) Specifications

Angle Modulators Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Frequency Modulators		2 / 4			2 / 4			1 per output channel
Phase Modulators		2 / 4			2 / 4			1 per output channel
Carrier Signal Source		FGs			FGs			Section 1.2.1 on page 14
Modulating Signal Source		AWGs			AWGs			Section 3.2.3.21 on page 59
Frequency Modulators (16-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	MHz	
Modulating Signal Resolution		16			16		Bits	AWG Waveform
Modulating Signal BW	0		250	0		500	MHz	AWG Nyquist limit
Deviation Gain	0		200	0		400	MHz	
Deviation Gain Resolution		16			16		Bits	
Frequency Modulators (32-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	MHz	
Modulating Signal Resolution		32			32		Bits	AWG Waveform
Modulating Signal BW	0		50	0		100	MHz	AWG Nyquist limit
Deviation Gain	0		200	0		400	MHz	
Deviation Gain Resolution		16			16		Bits	
Phase Modulators (16-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	Deg	
Modulating Signal Resolution		16			16		Bits	AWG Waveform
Modulating Signal BW	0		250	0		500	MHz	AWG Nyquist limit
Deviation Gain	0		180	0		180	Deg	
Deviation Gain Resolution		16			16		Bits	$\sim$ 5.5 mdeg
Phase Modulators (32-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	Deg	
Modulating Signal Resolution		16			16		Bits	AWG Waveform is truncated
Modulating Signal BW	0		50	0		100	MHz	AWG Nyquist limit
Deviation Gain	0		180	0		180	Deg	
Deviation Gain Resolution		16			16		Bits	$\sim$ 5.5 mdeg

Table 8: Angle Modulators Specifications

Amplitude Modulators Specifications

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Amplitude Modulators		2 / 4			2 / 4			1 per output channel
Offset Modulators		2 / 4			2 / 4			1 per output channel
Carrier Signal Source		FGs			FGs			Section 1.2.1 on page 14
Modulating Signal Source		AWGs			AWGs			Section 3.2.3.21 on page 59
Amplitude & Offset Modulators (16-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	V_p	
Modulating Signal Resolution		16			16		Bits	AWG Waveform
Modulating Signal BW	0		250	0		500	MHz	AWG Nyquist limit
Deviation Gain	0		1.5	0		1	V_p	
Deviation Gain Resolution		16			14		Bits	Limited by the output DAC
Amplitude & Offset Modulators (32-bit Modulating Waveform)								
Deviation	-Dev. Gain		+Dev. Gain	-Dev. Gain		+Dev. Gain	V_p	
Modulating Signal Resolution		16			16		Bits	AWG Waveform is truncated
Modulating Signal BW	0		50	0		100	MHz	AWG Nyquist limit
Deviation Gain	0		1.5	0		1	V_p	
Deviation Gain Resolution		16			14		Bits	Limited by the output DAC

Table 9: Amplitude Modulators Specifications

Clock System Specifications

	AWG-H3350(F)			AWG-H3380(F)			AWG-H3340			AWG-H3330				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications														
Clock Frequency	>100		500	>100		1000		500			1000		MHz	

Table 10: Clock System Specifications

AC Performance

	AWG-H3350(F) / H3330			AWG-H3380(F) / H3340				
Parameter	**Min**	**Typ**	**Max**	**Min**	**Typ**	**Max**	**Units**	**Comments**
General Specifications								
Analog Output Jitter			<2			<2	ps	RMS (cycle-to-cycle)
AWG Trigger to Output Jitter			<2			<2	ps	RMS (cycle-to-cycle). For any trigger referenced to the chassis clock. Independent of input trigger jitter if input jitter < 4nS peak-to-peak
Trigger Resolution		10			10		ns	
Channel-to-channel Skew			<20			<50	ps	Between ch 0 & ch 1, and ch 2 & ch 3
			<50			<50	ps	Between any channel
			<150			<150	ps	Between modules
Clock Output Jitter			<2			<2	ps	RMS (cycle-to-cycle)
Clock Accuracy and Stability			100			100	ppm	(PXIe, cPCIe Versions). Chassis dependent. This value corresponds to a chassis that fulfils the PXI Express specifications. This value can be improved with an external chassis clock or a System Timing Module.
AC Specifications								
SFDR (Spurious-Free Dynamic Range)								P_{out} = 4 dBm (AWG-H3350(F)/H3330) and 0 dBm (AWG-H3380(F)/H3340). Measured from DC to max frequency
f_{out} = 10 MHz	60	63		60	64		dBc	
f_{out} = 40 MHz	60	63		60	62		dBc	
f_{out} = 80 MHz	61	64		61	63		dBc	
f_{out} = 120 MHz	66	69		58	61		dBc	
f_{out} = 160 MHz	64	67		54	57		dBc	
f_{out} = 200 MHz	60	63		52	55		dBc	
f_{out} = 320 MHz	-	-		50	53		dBc	
f_{out} = 400 MHz	-	-		50	52		dBc	
Crosstalk (Adjacent Channels)								
f_{out} = 10 MHz		<-105	-105		<-105	-105	dB	
f_{out} = 40 MHz		-85	-82		-85	-82	dB	
f_{out} = 80 MHz		-75	-72		-80	-77	dB	
f_{out} = 120 MHz		-88	-85		-89	-86	dB	
f_{out} = 160 MHz		-73	-70		-76	-73	dB	
f_{out} = 200 MHz		-85	-82		-86	-83	dB	
f_{out} = 320 MHz		-	-		-83	-80	dB	
f_{out} = 400 MHz		-	-		-81	-78	dB	
Crosstalk (Non-adjacent Channels)								
f_{out} = 10 MHz		<-105	-105		<-105	-105	dB	
f_{out} = 40 MHz		-86	-83		-89	-86	dB	
f_{out} = 80 MHz		-78	-75		-81	-78	dB	
f_{out} = 120 MHz		<-105	-105		-103	-100	dB	
f_{out} = 160 MHz		-92	-89		-95	-92	dB	
f_{out} = 200 MHz		-100	-97		-102	-99	dB	
f_{out} = 320 MHz		-	-		-97	-94	dB	
f_{out} = 400 MHz		-	-		-95	-92	dB	
Phase Noise (SSB)								
offset = 1 KHz			<-127			<-127	dBc/Hz	
offset = 10 KHz			<-133			<-133	dBc/Hz	
offset = 100 KHz			<-138			<-138	dBc/Hz	
Average Noise Power Density			<-144			<-144	dBm/Hz	

Table 11: AC Performance

Typical AC Performance (AWG-H3350(F) / H3330)

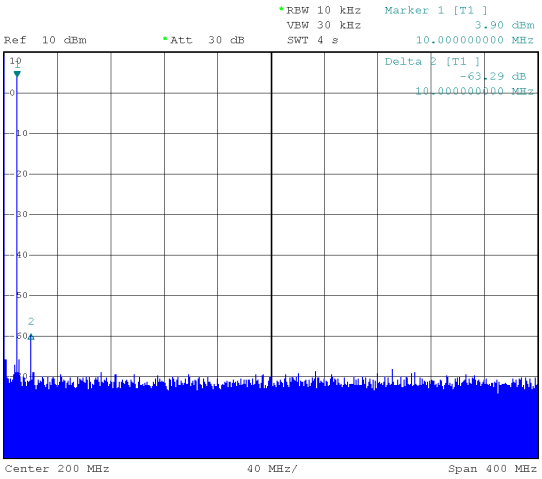


Figure 2: Single-tone spectrum @ f_{out} = 10 MHz

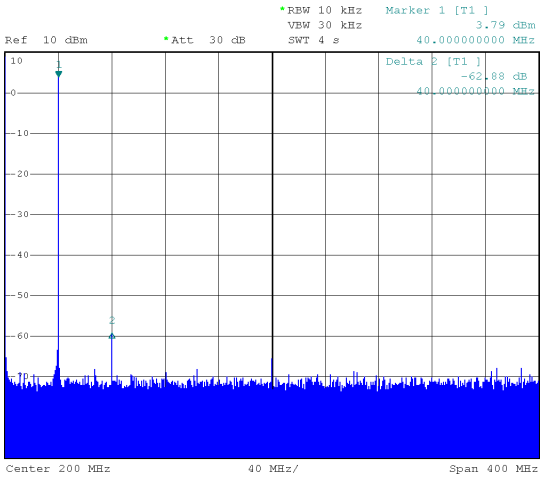


Figure 5: Single-tone spectrum @ f_{out} = 40 MHz

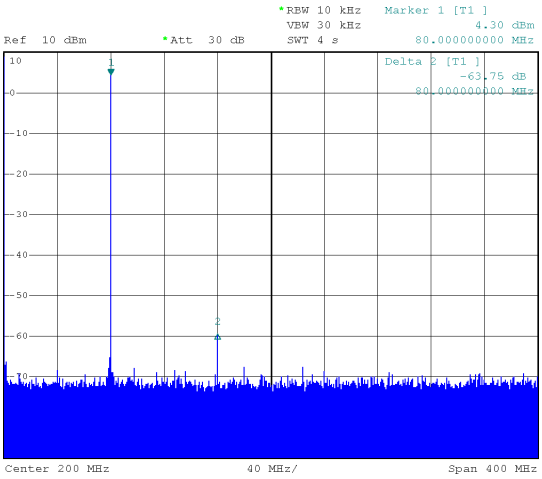


Figure 3: Single-tone spectrum @ f_{out} = 80 MHz

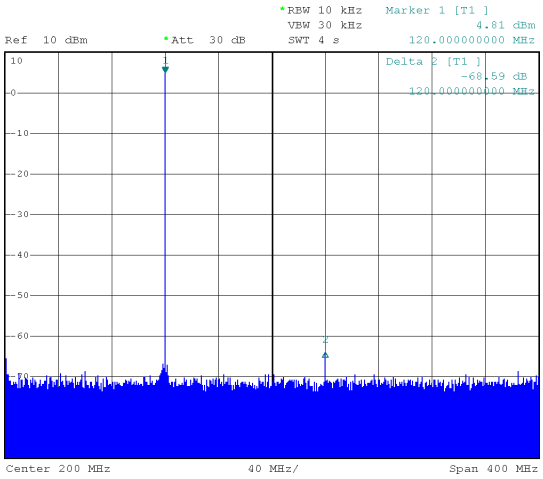


Figure 6: Single-tone spectrum @ f_{out} = 120 MHz

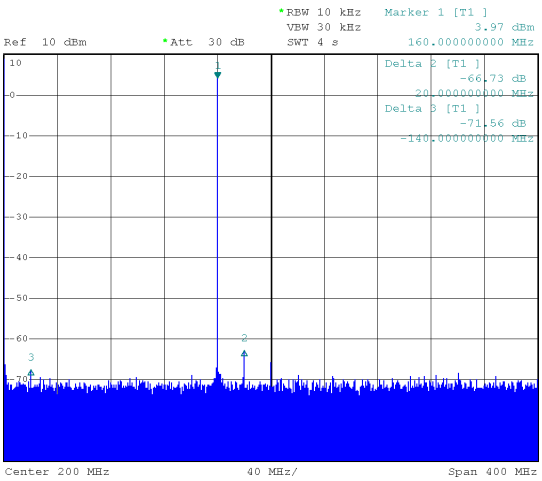


Figure 4: Single-tone spectrum @ f_{out} = 160 MHz

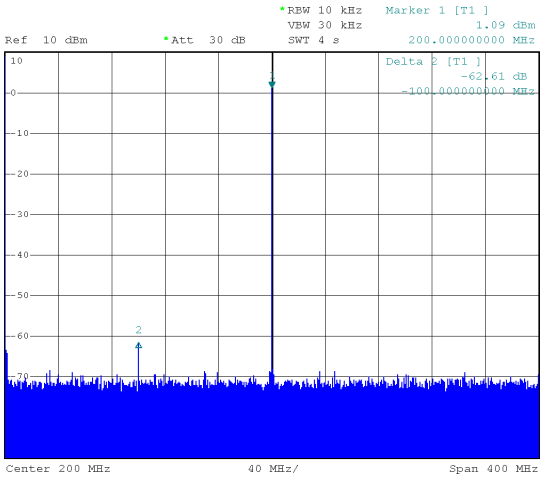


Figure 7: Single-tone spectrum @ f_{out} = 200 MHz

Typical AC Performance (AWG-H3380(F) / H3340)

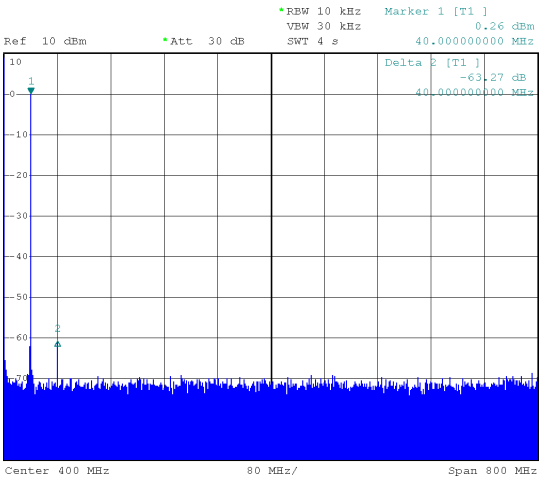


Figure 8: Single-tone spectrum @ f_{out} = 40 MHz

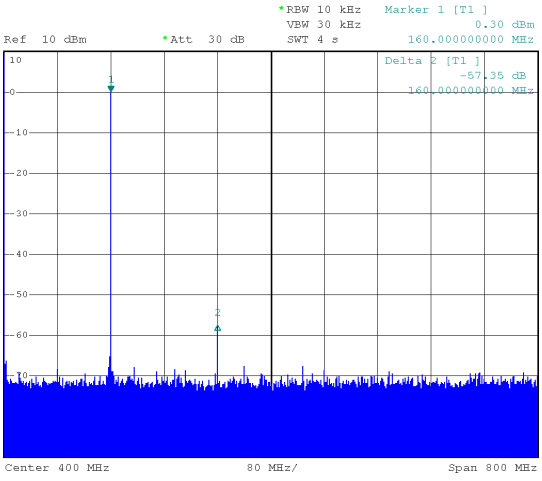


Figure 10: Single-tone spectrum @ f_{out} = 160 MHz

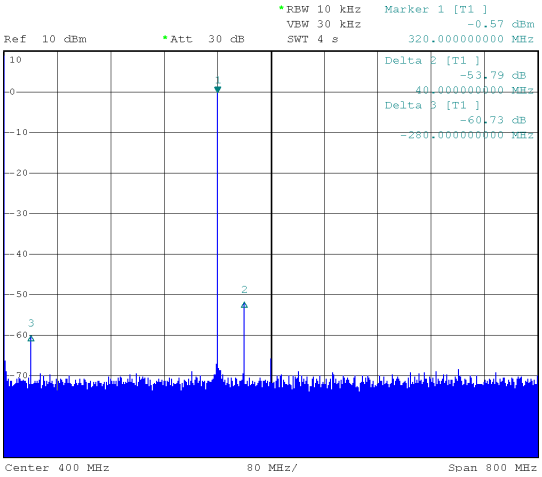


Figure 9: Single-tone spectrum @ f_{out} = 320 MHz

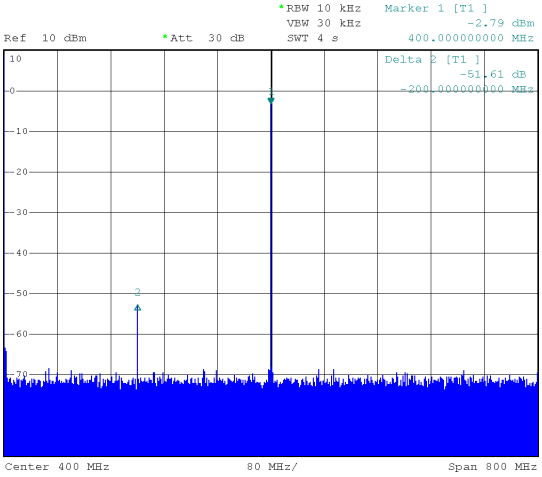


Figure 11: Single-tone spectrum @ f_{out} = 400 MHz

Contents

1 Theory of Operation: SD AWG-H3300/H3300F Series

1.1 Analog Outputs Overview

The SD AWG-H3300/H3300F Series has a very flexible and powerful output structure that allows the user to create complex waveforms.

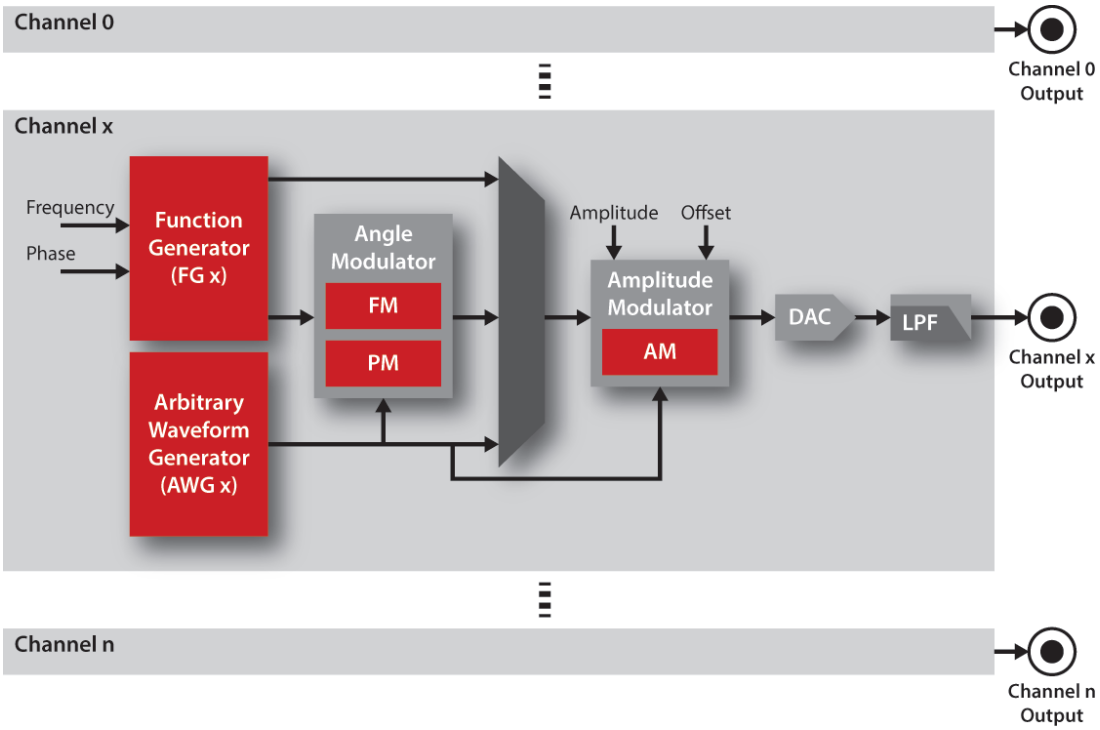


Figure 12: SD AWG-H3300/H3300F Series functional block diagram. All channels have identical output structure.

As shown in Figure 12, the SD AWG-H3300/H3300F Series includes embedded Arbitrary Waveform Generators (AWGs), Function Generators (FG) and modulators, converting the module into a powerful ready-to-use signal generator. The SD AWG-H3300/H3300F Series is capable of generating any standard waveform (sinusoidal, triangular, square, DC voltages) or any arbitrary waveform defined by the user and stored in its onboard RAM. With the embedded modulators, the output channels can be modulated in phase, frequency or amplitude in order to create any existing analog or digital modulation.

1.2 Signal Generation

Using the embedded Function Generators (Section 1.2.1 on the following page) and the Arbitrary Waveform Generators (1.2.2 on page 16), the SD AWG-H3300/H3300F Series is able to generate the following signals:

- **Basic periodic signals:** sinusoidal (RF), triangular, square, etc. They can be modulated in phase, frequency or amplitude.
- **Arbitrary waveforms:** which can be send directly to the output, or they can be used as a modulating signal for the modulators.
- **DC voltage**

The signal generation options are shown in Table 12. The selection is performed with the function *channelWaveShape*, Section 3.2.3.1 on page 39.

Signal Type

		Programming Definitions	
Option	**Description**	**Name**	**Value**
No Signal	The output signal is set to 0. All other channel settings are maintained	AOU_OFF (default)	-1
Sinusoidal	Generated by the Function Generator (Section 1.2.1 on the following page)	AOU_SINUSOIDAL	1
Triangular	Generated by the Function Generator (Section 1.2.1 on the next page)	AOU_TRIANGULAR	2
Square	Generated by the Function Generator (Section 1.2.1 on the following page)	AOU_SQUARE	4
DC Voltage	The output DC voltage is set by the channel amplitude setting (Section 1.2.3 on page 23)	AOU_DC	5
Arbitrary Waveform	Generated by the Arbitrary Waveform Generator (1.2.2 on page 16)	AOU_AWG	6
Partner Channel	Only for odd channels. It is the output of the previous channel (to create differential signals, etc.)	AOU_PARTNER	8

Table 12: Output signal selection (parameter *waveShape* in function *channelWaveShape*, Section 3.2.3.1 on page 39)

Programming Information

Function Name	Comments	Details
channelWaveShape	It selects the signal type	Section 3.2.3.1 on page 39

Table 13: Programming functions related to the signal selection

1.2.1 Function Generators (FGs)

Each output channel has a dedicated Function Generator (FG) which generates basic periodic signals. The FG is commonly used to generate the RF carrier in modulation schemes. FGs specifications are shown in Table 5 on page 4.

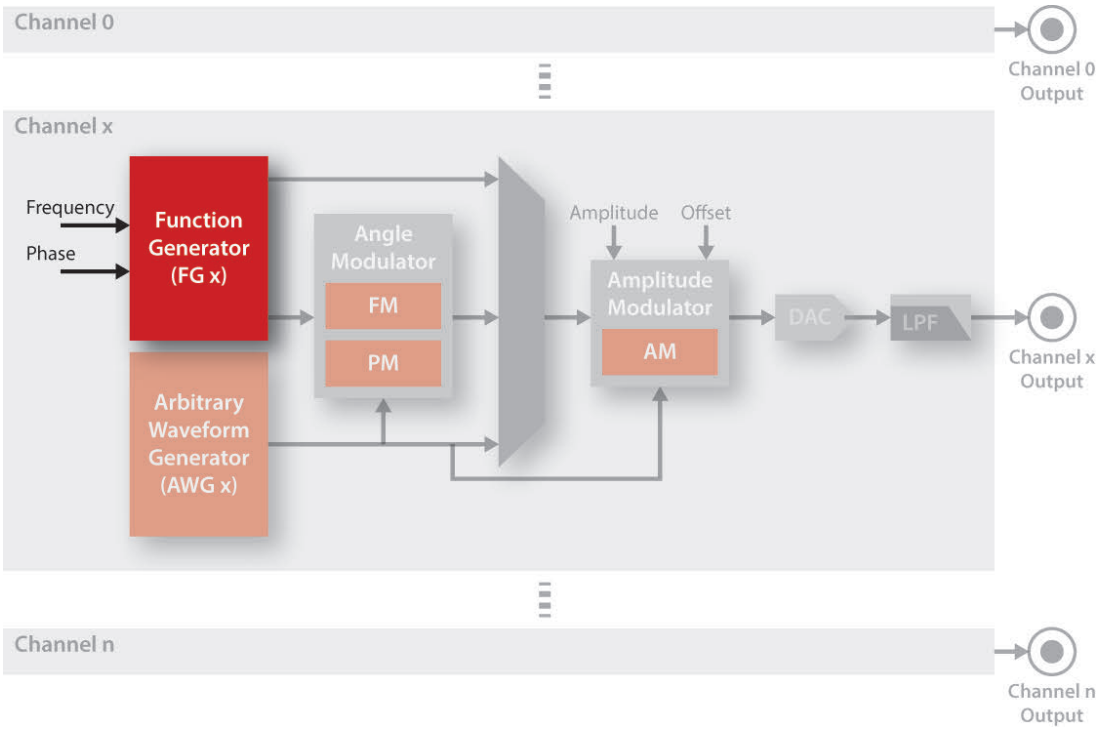


Figure 13: The Function Generator within the SD AWG-H3300/H3300F Series functional block diagram

NOTE

Waveform harmonics: Non-sinusoidal wave shapes (triangular, square, etc.) have high frequency components that may fall outside the bandwidth of the reconstruction filter if the fundamental frequency is too high. In this situation, the output analog signal may suffer some distortion due to the missing harmonics, becoming a sinusoidal as the fundamental frequency approaches the cutoff frequency of the filter.

ADVANCED

Phase coherent vs. phase continuous: Changes in the output signal are always phase continuous, not phase coherent. Therefore, when, for example, the frequency is changed from freq1 to freq2 and changed back to freq1, the phase will not be the initial one. To achieve phase coherent behaviour, the channel accumulated phase can be reset (function *channelPhaseReset*, Section 3.2.3.4 on page 42). In addition, in HVI operation (Section 2.1.1 on page 30) the execution time is deterministic, which allows the user to calculate the new phase and adjust it after any frequency change.

NOTE

FGs vs AWGs: When the generation of periodic signals is needed, an FG has many advantages over a pure AWG solution:

- The FG does not use onboard RAM.
- The waveshape, the frequency and the phase can be changed in real time without having to modify a static waveform loaded in memory.
- Achieving the same precision in frequency and phase with a pure AWG solution requires a huge amount of memory.

Programming Information

Function Name	Comments	Details
channelWaveShape	It selects the signal type	Section 3.2.3.1 on page 39
channelFrequency	It sets the frequency of the FG	Section 3.2.3.2 on page 40
channelPhase	It sets the phase of the FG	Section 3.2.3.3 on page 41
channelPhaseReset	It resets the accumulated phase	Section 3.2.3.4 on page 42

Table 14: Programming functions related to the Function Generators

1.2.2 Arbitrary Waveform Generators (AWGs)

Arbitrary waveforms can be generated using powerful and flexible Arbitrary Waveform Generator (AWG) blocks (Figure 14). AWG specifications are shown in Table 7 on page 5.

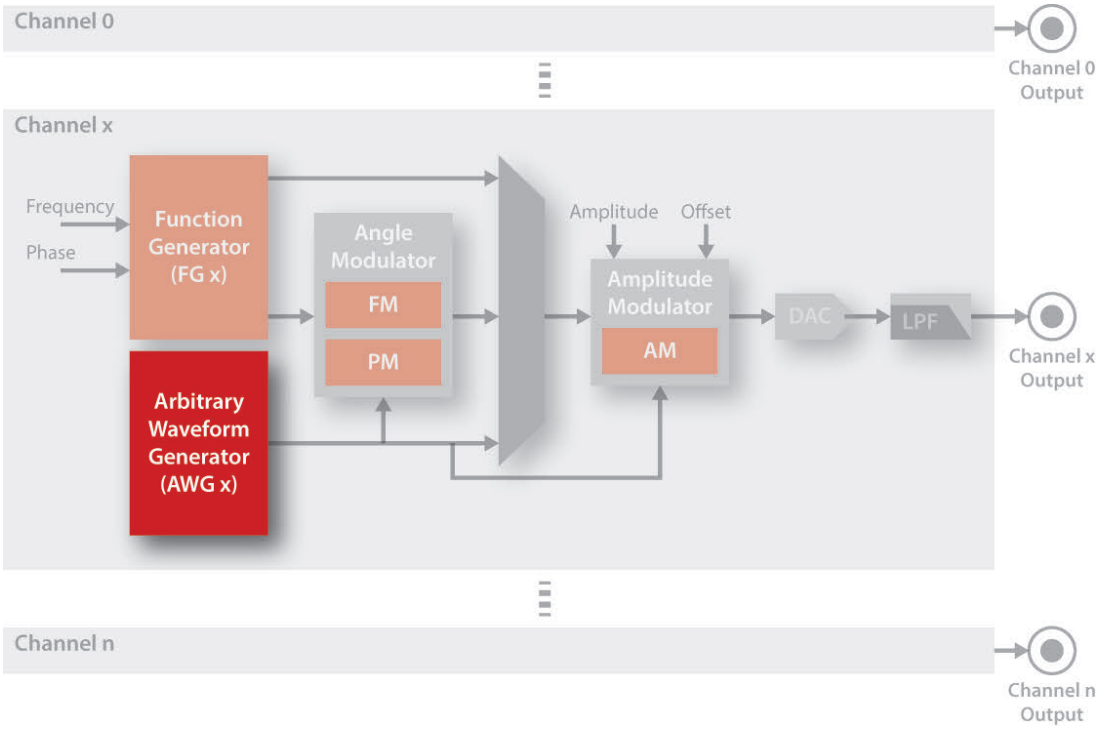


Figure 14: Arbitrary Waveform Generators (AWGs) in the SD AWG-H3300/H3300F Series

1.2.2.1 Introduction

Each AWG block has a flexible waveform queue system that allows the user to configure complex generation sequences. In order to generate waveforms, they must be loaded into the module onboard RAM, and they must be queued in the corresponding AWG.

The AWG queue system has the following advantages:

- It provides a way to generate a sequence of waveforms one after the other with no discontinuities.
- It allows the user to select many parameters (trigger mode, start delay, cycles, prescaler, etc.) individually per queued waveform.
- Waveforms can have many instances in many AWG queues, but only one copy is required in the onboard RAM. This feature saves onboard RAM memory.

1.2.2.2 Operation

Arbitrary waveform generation can performed using two methods:

1. **One-step programming**: The function *AWG*, Section 3.2.3.21 on page 59, provides a one-step solution to load, queue and run a single waveform directly from a file or from an array in the PC. This method simplifies the generation of a single waveform, but it does not allow the user to control important aspects of the waveform generation:
   (a) The possibility to prepare the AWG queue with multiple waveforms in advance before starting the generation. This is important to create more complex generation sequences (Figure 16 on the facing page).
   (b) The possibility to have a variety of waveforms in the onboard RAM in order to queue them repeatedly in the AWGs in an efficient way.
   (c) The precise moment when waveforms are transferred from a file to the PC RAM and to the onboard RAM. This may be important for long waveforms and time-critical applications.
2. **Step-by-step programming:** Signadyne Programming Libraries provide full control of all the aspects of the arbitrary waveform generation (Figure 15 on the next page). Instead of using the function *AWG*, Section 3.2.3.21 on page 59, the step-by-step procedure is performed as follows:
   (a) Create waveforms in the PC RAM with SD-Wave Class (function *new*, Section 3.2.5.1 on page 90). Waveforms can be created from points in an array or in a hard disk file.

(b) Transfer the waveforms to the module onboard RAM (function *waveformLoad*, Section 3.2.3.18 on page 56).
(c) Queue the waveforms in any AWG to create the desired generation sequence using the function *AWGqueueWaveform*, Section 3.2.3.22 on page 61 (See AWG Queue System).
(d) Start the generation with function *AWGstart*, Section 3.2.3.24 on page 63, and provide triggers if required (See AWG Queue System).

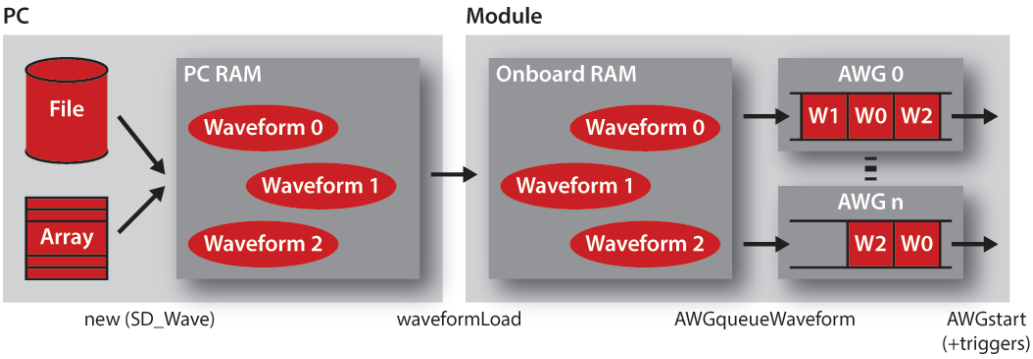


Figure 15: Step-by-step process to configure arbitrary waveforms

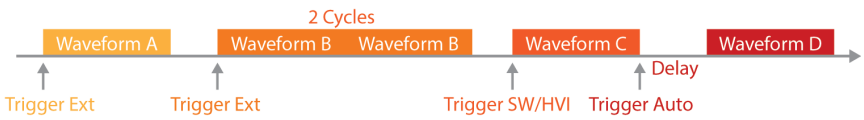
1.2.2.3 Queue System

Figure 16 shows some generation examples using the AWG queue system. Each queued waveform has the following parameters:

- **Trigger Mode:** it selects the trigger for the waveform (Table 15 on the following page).
- **Start Delay:** an optional delay from the reception of the trigger to the beginning of the waveform generation.
- **Cycles:** the number of times the waveform will be repeated. Using the right trigger mode, the AWG can be set to require one trigger per cycle or just one trigger at the beginning of all the cycles.
- **Prescaler:** it divides the effective waveform sampling rate (Equation 1 on the next page).

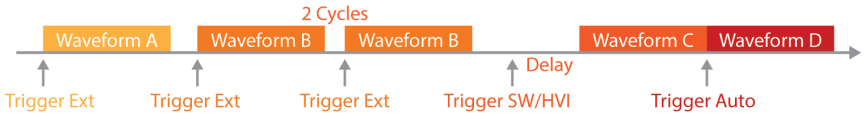
Queue

Waveform	Trigger	Delay	Cycles	Prescaler
A	Ext	0	1	0
B	Ext	0	2	0
C	SW/HVI	0	1	0
D	Auto	100	1	0



Queue

Waveform	Trigger	Delay	Cycles	Prescaler
A	Ext	0	1	0
B	Ext (cycle)	0	2	0
C	SW/HVI	100	1	0
D	Auto	0	1	0



Queue

Waveform	Trigger	Delay	Cycles	Prescaler
A	Ext	0	1	0
B	Auto	0	1	0
C	SW/HVI	0	1	1
D	Ext	0	0	0

Figure 16: Examples of the AWG queue system

NOTE
Inter-waveform/inter-cycles Discontinuities: The queue system of the Arbitrary Waveform Generators (AWGs) allows the user to queue many waveforms one after the other. It also provides the capability to set repetition cycles per waveform and repetition cycles for the complete queue. In all these cases there are absolutely no discontinuities between waveforms or cycles, providing a continuous waveform generation from the last sample point of the finished waveform to the first sample point of the starting waveform.

TIP
HVI Generation Sequences: The queue system of the Arbitrary Waveform Generators (AWGs) provides a way to create generation sequences. For more complex sequences, the best solution is the use of Hard Virtual Instruments (HVIs), which are time deterministic sequences with nanosecond resolution, providing a hard real-time execution (Section 2.1.1 on page 30). HVIs are programmed with Signadyne ProcessFlow [1], a user-friendly flowchart-style programming environment.

1.2.2.4 Prescaler and Sampling Rate

As mentioned in Section 1.2.2.3 on the previous page, the user can set a different prescaler value for each of the waveforms queued in the AWG. This prescaler reduces the effective sampling frequency as follows:

$$\begin{cases} f_s = f_{CLKsys} & prescaler = 0 \\ f_s = \frac{f_{CLKsys}}{5 \cdot prescaler} & prescaler > 0 \end{cases} \quad (1)$$

where:

- fs is final effective sampling frequency.
- f_{CLKsys} is the system clock frequency (Section 1.4 on page 27).
- $prescaler$ is an integer value (0..4095).

An important advantage of this method is the possibility to change the sampling frequency in real time from one waveform to another, reducing waveform sizes and maximizing the flexibility of the AWG.

ADVANCED
Prescaler and aliasing: Note that reducing the effective sampling rate produces aliasing inside the reconstruction filter bandwidth. Therefore, this unfiltered alias appears at the output.

1.2.2.5 Trigger

As previously explained, the user can configure a different trigger mode in each queued waveform. The available trigger modes are show in Table 15).

AWG Trigger Mode

		Programming Definitions	
Option	**Description**	**Name**	**Value**
Auto	The waveform is launched automatically after function *AWGstart*, Section 3.2.3.24 on page 63, or when the previous waveform in the queue finishes	AUTOTRIG	0
SW / HVI	Software trigger. The AWG is triggered by the function *AWGtrigger* (Section 3.2.3.37 on page 76) provided that the AWG is running. *AWGtrigger* can be executed from a VI or an HVI (see HVI and VI details, Section 2.1.1 on page 30)	SWHVITRIG[1]	1
SW / HVI (per cycle)	Software trigger. Identical to the previous option, but the trigger is required per each waveform cycle	SWHVITRIG_CYCLE	5
External Trigger	Hardware trigger. The AWG waits for an external trigger (Table 16 on the next page)	EXTTRIG	2
External Trigger (per cycle)	Hardware trigger. Identical to the previous option, but the trigger is required per each waveform cycle	EXTTRIG_CYCLE	6

[1] VIHVITRIG is equivalent but is considered obsolete

Table 15: Trigger methods for the waveforms queued in an AWG (parameter triggerMode in function *AWG*, Section 3.2.3.21 on page 59, or in function *AWGqueueWaveform*, Section 3.2.3.22 on page 61)

If the queued waveforms are going to use any of the External Trigger modes, the source of this trigger must be configured using the function *AWGtriggerExternalConfig*, Section 3.2.3.36 on page 75. The available external trigger options are shown in Table 16 on the next page and Table 17 on the facing page.

NOTE
External Trigger Connector/Line Usage: Apart from the AWG trigger settings, an external trigger connector/line may have additional settings (input/output direction, sampling/synchronization options, etc.) that need to be configured for proper operation (Section 1.5 on page 28).

AWG External Trigger Source

Option	Description	Programming Definitions	
		Name	Value
External Trigger	The AWG external trigger is the TRG connector/line of the module. PXI form factor only: this trigger can be synchronized to CLK10, see Table 28 on page 28	TRIG_EXTERNAL	0
PXI Trigger [0..n]	PXI form factor only. The AWG external trigger is a PXI trigger line and it is synchronized to CLK10	TRIG_PXI + Trigger No.	4000 + Trigger No.

Table 16: External trigger source for the AWGs (parameter *externalSource* in function *AWGtriggerExternalConfig*, Section 3.2.3.36 on page 75)

AWG External Trigger Behaviour

Option	Description	Programming Definitions	
		Name	Value
Active High	Trigger is active when it is at level high	TRIG_HIGH	1
Active Low	Trigger is active when it is at level Low	TRIG_LOW	2
Rising Edge	Trigger is active on the rising edge	TRIG_RISE	3
Falling Edge	Trigger is active on the falling edge	TRIG_FALL	4

Table 17: External trigger behaviour for the AWGs (parameter *triggerBehaviour* in function *AWGtriggerExternalConfig*, Section 3.2.3.36 on page 75)

1.2.2.6 FlexCLK Synchronization (models with variable sampling rate)

Section 1.4.1 on page 27 shows the internal diagram and the operation of the SD AWG-H3300/H3300F SeriesFlexCLK system. This advanced technology allows the user to change the sampling frequency of the AWGs (CLKsys), while maintaining full synchronization capabilities thanks to the internal CLKsync signal.

CLKsync is an internal signal used to start the AWGs, and it is aligned with CLKsys and PXI_CLK10. Its frequency depends on the CLKsys frequency (Section 1.4.1 on page 27), giving as a result the following scenarios:

- $f_{CLKsync} = f_{PXI\_CLK10}$

  When both frequencies coincide, there is no phase uncertainty between both signals, and a trigger synchronized with PXI_CLK10 will start the AWGs always with the same skew (Figure 17), independently of the clock boot conditions. This ensures proper synchronization also between different modules.

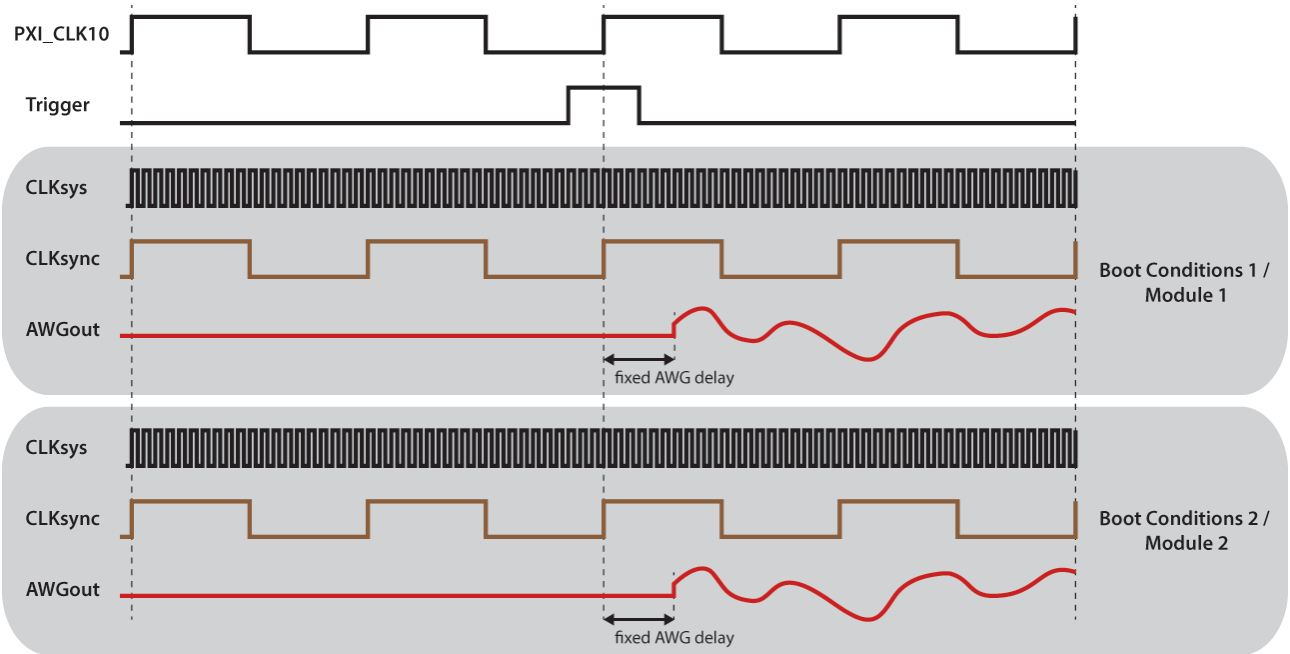


Figure 17

- $f_{CLKsync} != f_{PXI\_CLK10}$

When both frequencies do not coincide, both signals are still aligned but there is a phase uncertainty due to their frequency difference. In this case, if a trigger synchronized with PXI_CLK10 is sent to the system, there might be a skew between the start of different AWGs (Figure 18).

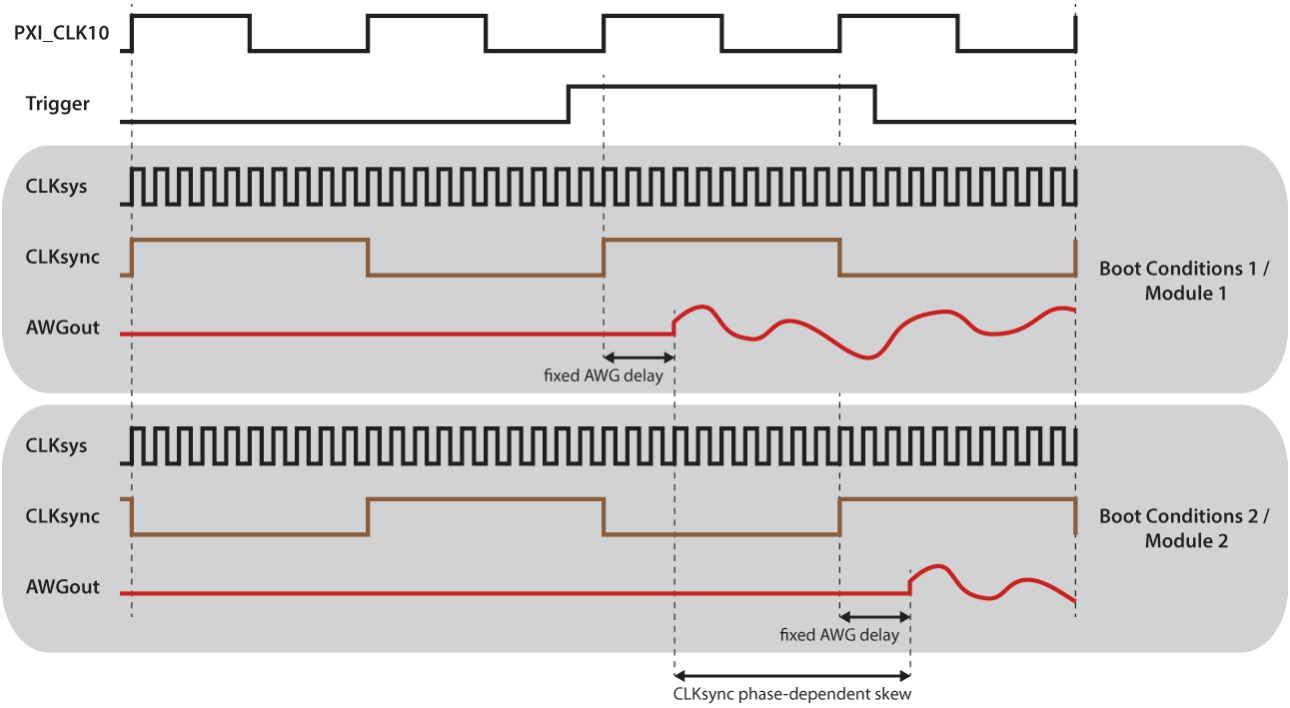


Figure 18

This phase uncertainty can be easily corrected using the function *clockResetPhase*, Section 3.2.3.17 on page 55. This functions sets the module/s in a sync mode, and the next trigger will be used to reset the phase of the CLKsync and CLKsys signals, not to start the AWGsIn this way the phase uncertainty between different modules or between different boot conditions can be eliminated, giving as a result a predictable and repeatable skew (Figure 19).

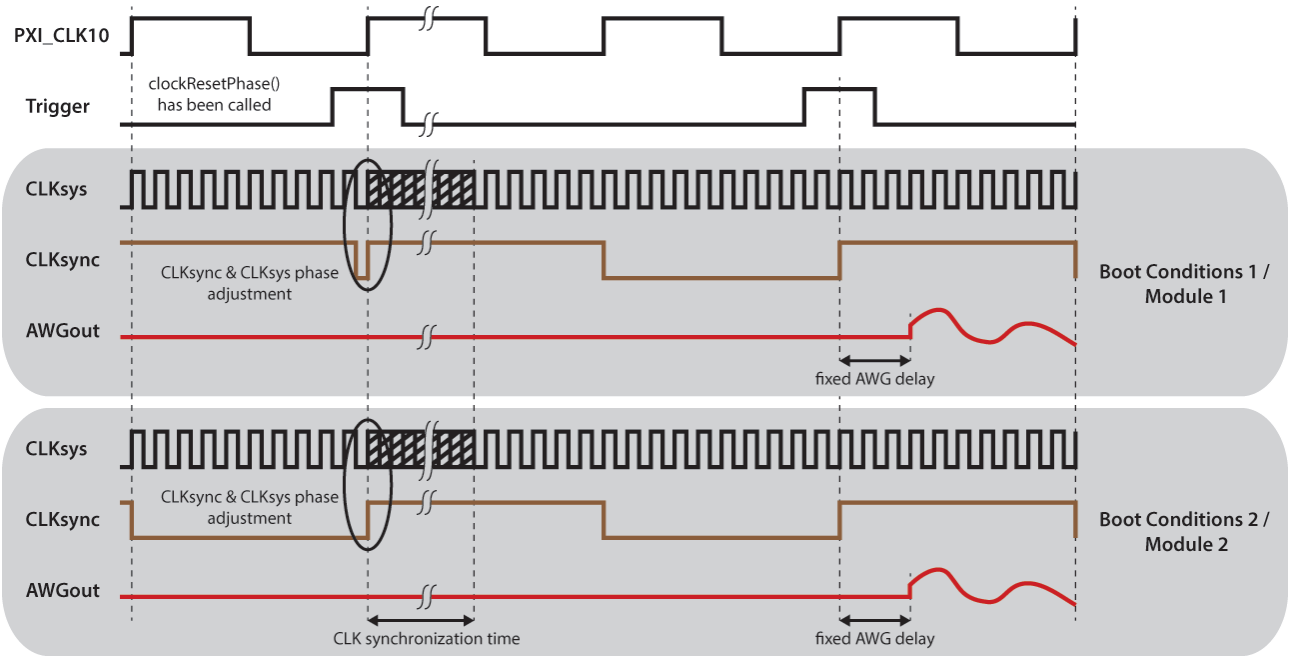


Figure 19

1.2.2.7 Waveform Array/File Estructure

As seen previously, the two posible sources of waveforms are an array in the PC RAM and a file in the PC HDD. The memory array is just an array of waveform points, without header and without any particular structure. On the other hand, a waveform file is a text file with values separated by comas (*.csv), with the structure shown in Figure 20.

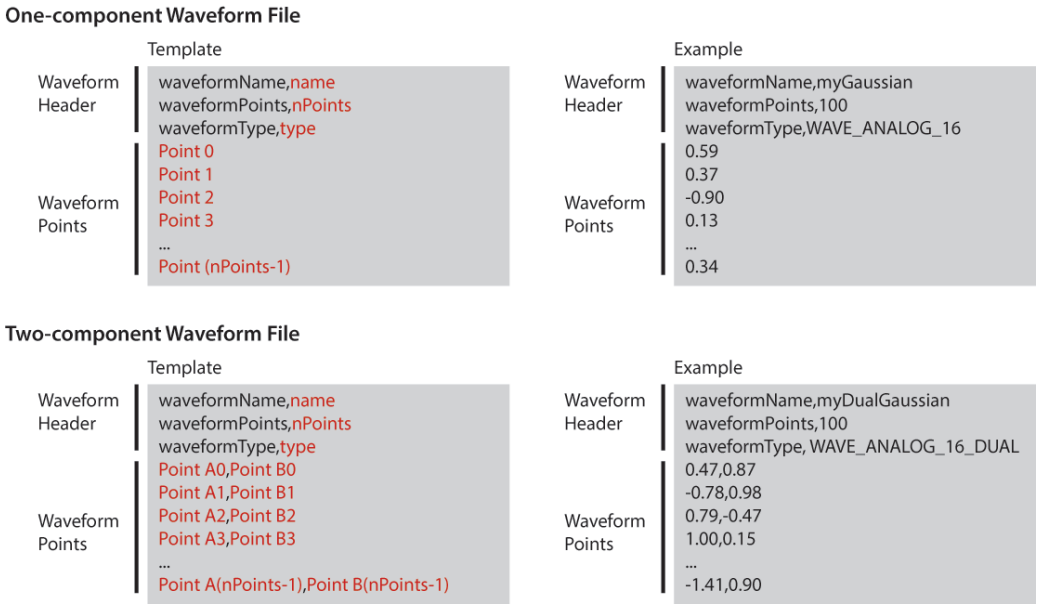


Figure 20: Waveform *.csv file structure

The waveform type is a parameter that tells the module which kind of waveform, and it used to configure internally the AWG. The waveform types available are shown in Table 18.

AWG Waveform types

		Programming Definitions	
Option	**Description**	**Name**	**Value**
Analog 16 Bits	Analog normalized waveforms (-1..1) defined with doubles	WAVE_ANALOG_16	0
Analog 32 Bits	Analog normalized waveforms (-1..1) defined with doubles	WAVE_ANALOG_32	1
Analog 16 Bits Dual	Analog normalized waveforms (-1..1) defined with doubles, with two components (A and B)	WAVE_ANALOG_16_DUAL	4
Analog 32 Bits Dual	Analog normalized waveforms (-1..1) defined with doubles, with two components (A and B)	WAVE_ANALOG_32_DUAL	6
IQ	Analog normalized waveforms (-1..1) defined with doubles, with two components (I and Q)	WAVE_IQ	2
IQ Polar	Analog waveforms (-1..1 module, -180..+180 phase) defined with doubles, with two components (Module and Phase)	WAVE_IQPOLAR	3
Digital	Digital waveforms defined with integers	WAVE_DIGITAL	5

Table 18: Waveform types for the AWGs (parameter *waveformType* in the waveform file or in function *new*, Section 3.2.5.1 on page 90)

1.2.2.8 Programming Functions Summary

Programming Information

Function Name	Comments	Details
waveformLoad	It loads a waveform into the module onboard RAM	Section 3.2.3.18 on page 56
waveformFlush	It deletes all the waveforms from the module onboard RAM and flushes all the AWG queues	Section 3.2.3.20 on page 58
AWG	It provides a one-step method to load, queue and start a single waveform	Section 3.2.3.21 on page 59
AWGqueueWaveform	It queues a waveform in the specified AWG	Section 3.2.3.22 on page 61
AWGflush	It empties the queue of the AWG	Section 3.2.3.23 on page 62
AWGstart	It runs the AWG starting from the beginning of its queue	Section 3.2.3.24 on page 63
AWGstartMultiple	Same as the previous one, but acting on more than one AWG at once	Section 3.2.3.25 on page 64
AWGpause	It pauses the AWG	Section 3.2.3.26 on page 65
AWGpauseMultiple	Same as the previous one, but acting on more than one AWG at once	Section 3.2.3.27 on page 66
AWGresume	It resumes the AWG	Section 3.2.3.28 on page 67
AWGresumeMultiple	Same as the previous one, but acting on more than one AWG at once	Section 3.2.3.29 on page 68
AWGstop	It stops the AWG, resetting the queue to its initial position	Section 3.2.3.30 on page 69
AWGstopMultiple	Same as the previous one, but acting on more than one AWG at once	Section 3.2.3.31 on page 70
AWGtriggerExternalConfig	It configures the external triggers of the AWG	Section 3.2.3.36 on page 75
AWGtrigger	It triggers the AWG	Section 3.2.3.37 on page 76
AWGtriggerMultiple	Same as the previous one, but acting on more than one AWG at once	Section 3.2.3.38 on page 77
new (SD_Wave)	It creates a new waveform object in the PC RAM from a file or from an array	Section 3.2.5.1 on page 90
delete (SD_Wave)	It deletes the waveform object from the PC RAM	Section 3.2.5.2 on page 91

Table 19: Programming functions related to the Arbitrary Waveform Generators (AWGs)

1.2.3 Amplitude and DC Offset Settings

Each output channel in the SD AWG-H3300/H3300F Series has an amplitude and offset control as shown in Figure 21. Amplitude and DC Offset specifications are shown in Table 6 on page 5.

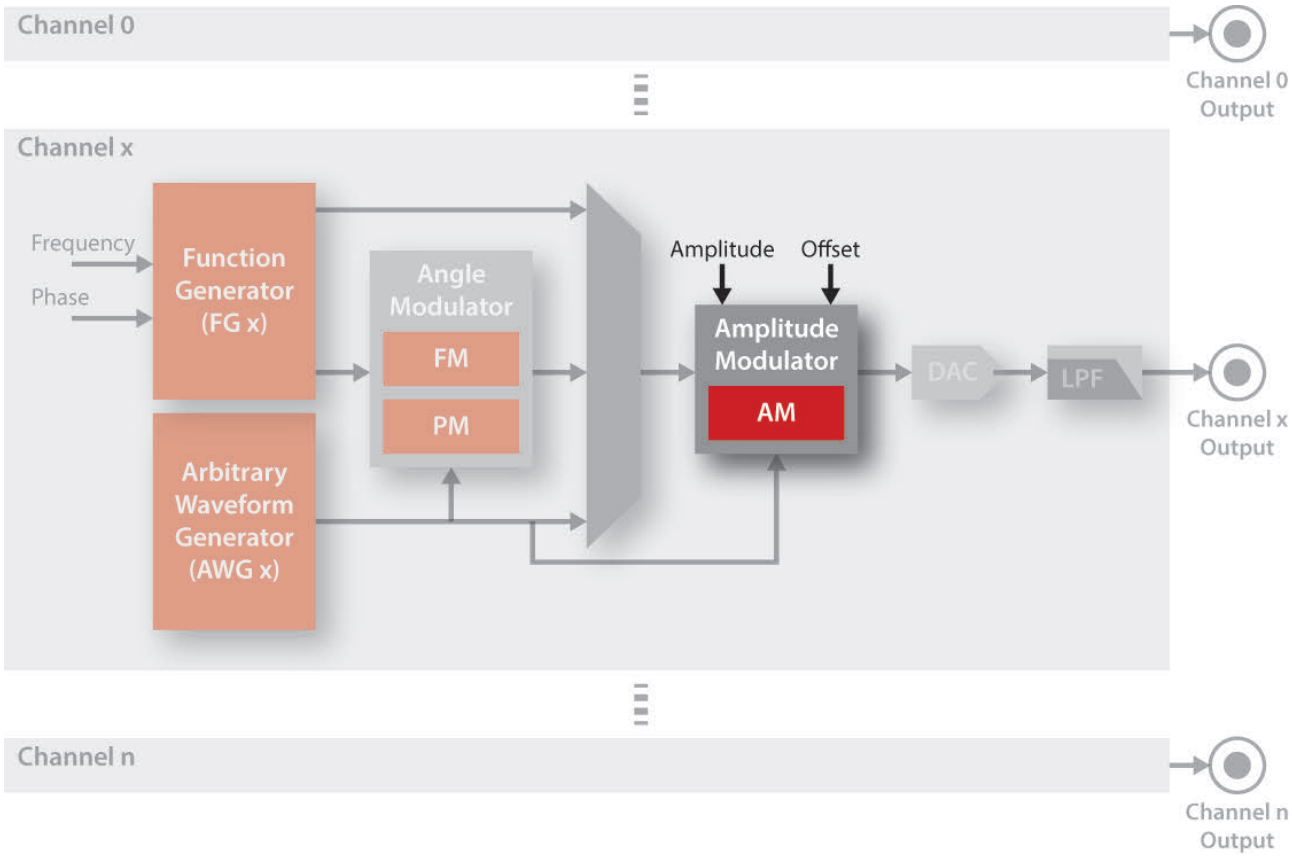


Figure 21: Amplitude and offset control within the SD AWG-H3300/H3300F Series functional block diagram

Programming Information

Function Name	Comments	Details
channelAmplitude	It sets the output amplitude	Section 3.2.3.6 on page 44
channelOffset	It sets the output DC offset	Section 3.2.3.7 on page 45

Table 20: Programming functions related to the amplitude and DC offset control

1.3 Signal Modulation

1.3.1 Angle Modulation (Frequency and Phase)

The output signal of the Function Generator (Section 1.2.1 on page 14) can be modulated in frequency or in phase using the angle modulator as shown in Figure 22. Angle modulation specifications are shown in Table 8 on page 6

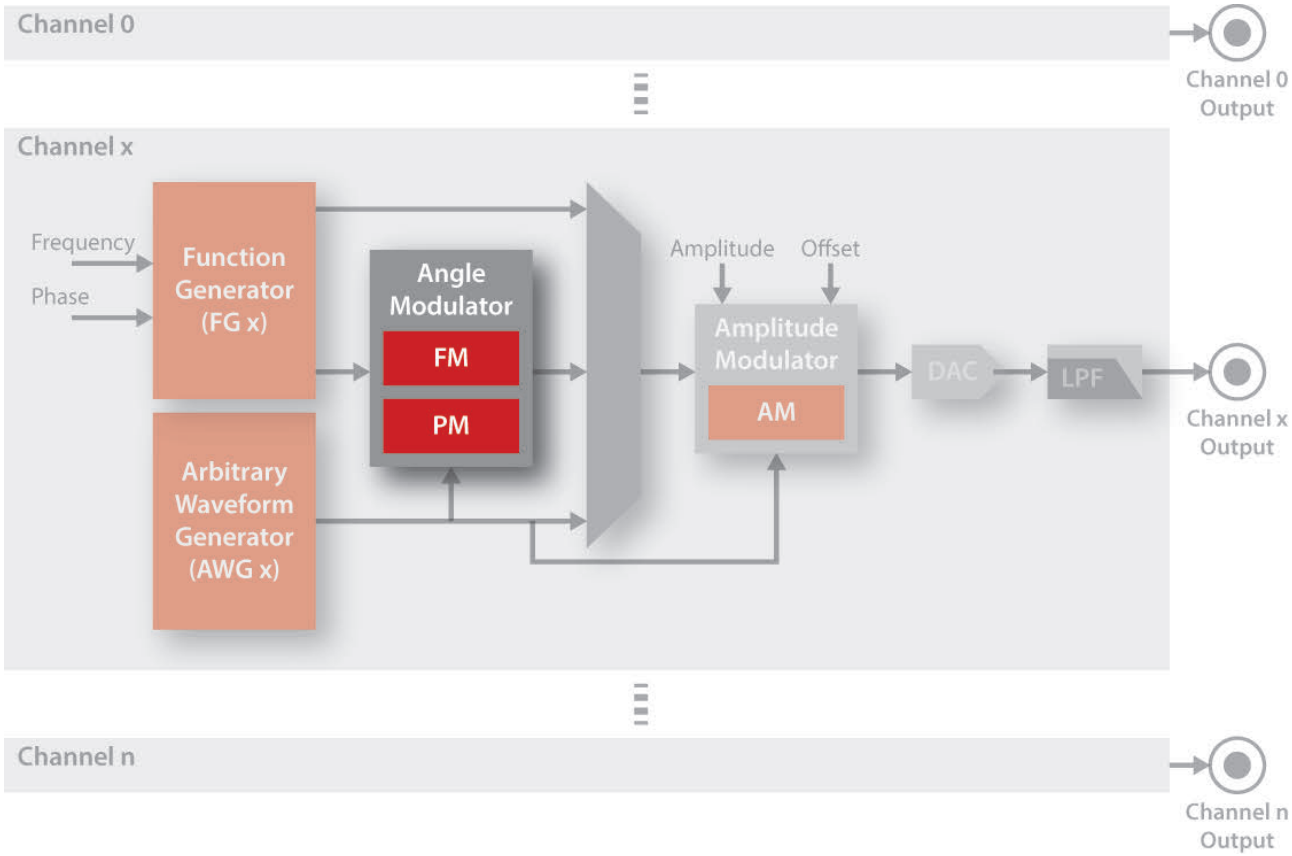


Figure 22: The angle modulator within the SD AWG-H3300/H3300F Series functional block diagram

Angle Modulation Types

Option	Description	Programming Definitions Name	Value
No Modulation	Modulation is disabled	AOU_MOD_OFF (default)	0
Frequency Modulation	The AWG is used to modulate the channel frequency	AOU_MOD_FM	1
Frequency Modulation (32 bits)[1]	The AWG is used to modulate the channel frequency	AOU_MOD_FM_32b	1
Phase Modulation	The AWG is used to modulate the channel phase	AOU_MOD_PM	2

[1] Models with 32-bit waveforms only

Table 21: Angle modulation options (parameter *modulationType* in function *modulationAngleConfig*, Section 3.2.3.8 on page 46)

The modulating signal is generated using the Arbitrary Waveform Generator (AWG, Section 1.2.2 on page 16) associated to that particular channel, e.g. AWG0 for channel 0. The angle modulator allows the user to create any analog/digital frequency or phase modulation, e.g. FM, FSK, PM, PSK, DPSK, etc.

The output signal of a channel using the frequency modulation is the following:

$$Output(t) = A \cdot \cos[2\pi(f_c + G \cdot AWG(t)) \cdot t + \phi_c] \quad (2)$$

where:

- A is the channel amplitude (set by the function *channelAmplitude*, Section 3.2.3.6 on page 44)
- G is the deviation gain or peak frequency deviation (set by *modulationAngleConfig*, Section 3.2.3.8 on page 46). G is only used for 16-bit waveforms.
- $AWG(t)$ is the normalized modulating signal generated by the AWG (Section 1.2.2 on page 16)

- $\cos(2\pi f_c t + \phi_c)$ is the carrier signal, generated with the Function Generator (Section 1.2.1 on page 14)

As an example, for the generation of an FM signal with an amplitude of 0.8 V_p, a modulation index (h) equal to 0.5 and a maximum frequency of the modulating signal of 10 MHz, the settings must be A=0.8 and G=5,000,000 ($G = h \cdot maxfreq[AWG(t)]$).

The output signal of a channel using the phase modulation is the following:

$$Output(t) = A \cdot \cos(2\pi f_c t + \phi_c + G \cdot AWG(t)) \quad (3)$$

where:

- A is the channel amplitude (set by the function *channelAmplitude*, Section 3.2.3.6 on page 44)
- G is the deviation gain or peak phase deviation (set by *modulationAngleConfig*, Section 3.2.3.8 on page 46). G is only used for 16-bit waveforms.
- $AWG(t)$ is the normalized modulating signal generated by the AWG (Section 1.2.2 on page 16)
- $\cos(2\pi f_c t + \phi_c)$ is the carrier signal, generated with the Function Generator (Section 1.2.1 on page 14)

As an example, for the generation of a PM signal with an amplitude of 0.8 V_p and a modulation index (h) equal to 180^0, the settings must be A=0.8 and G=180 (G=h).

Programming Information

Function	Comments	Details
modulationAngleConfig	It configures the angle modulator	Section 3.2.3.8 on page 46
AWG functions	They control the modulating signal	Section 1.2.2 on page 16
FG functions	They control the carrier signal	Section 1.2.1 on page 14

Table 22: Programming functions related to the angle modulator

1.3.2 Amplitude Modulation

The amplitude modulator can be used to modulate the amplitude of the output signal as show in Figure 23. This modulator can be used also to change the DC offset of the output signal. Amplitude modulators specifications are shown in Table 9 on page 6.

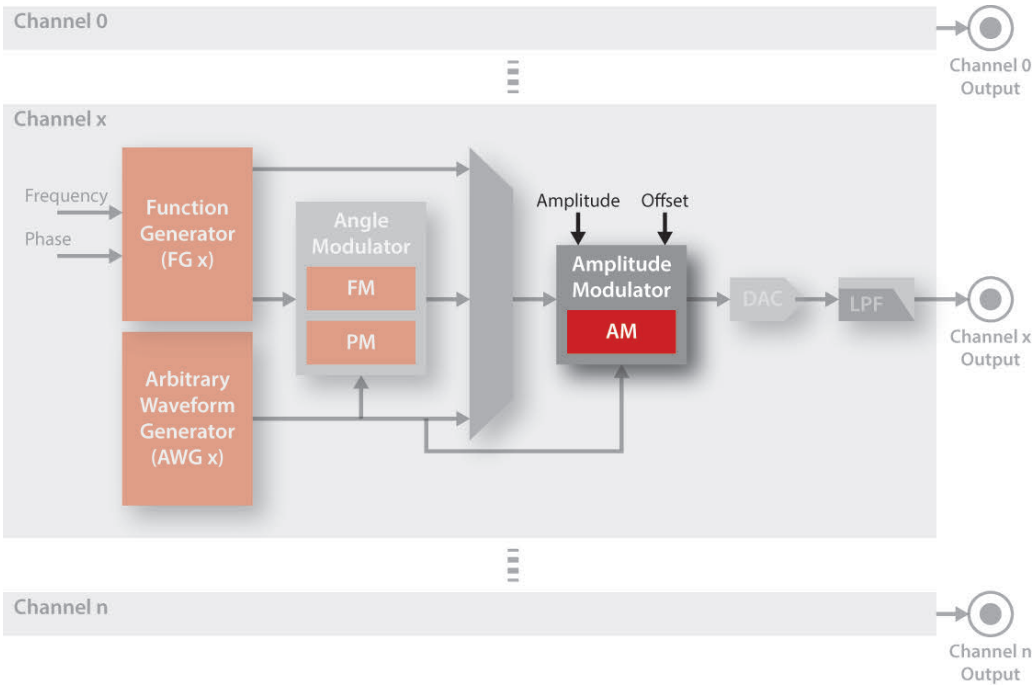


Figure 23: The amplitude modulator within the SD AWG-H3300/H3300F Series functional block diagram

The modulating signal is generated using the Arbitrary Waveform Generator (AWG, Section 1.2.2 on page 16) associated to that particular channel, e.g. AWG0 for channel 0. The amplitude modulator allows the user to create any analog/digital amplitude modulation, e.g. AM, ASK, etc.

The output signal of a channel using the amplitude modulator is the following:

$$Output(t) = (A + G \cdot AWG(t)) \cdot \cos(2\pi f_c t + \phi_c) \tag{4}$$

where:

- A is the channel amplitude (set by the function *channelAmplitude*, Section 3.2.3.6 on page 44)
- G is the deviation gain (set by the function *modulationAmplitudeConfig*, Section 3.2.3.9 on page 47). G is only used for 16-bit waveforms.
- $AWG(t)$ is the normalized modulating signal generated by the AWG (Section 1.2.2 on page 16)
- $\cos(2\pi f_c t + \phi_c)$ is the carrier signal, generated with the Function Generator (Section 1.2.1 on page 14)

As an example, for the generation of an AM signal with an amplitude of 0.8 V_p and a modulation index (h) equal to 0.5, the settings must be A=0.8 and G=0.32 ($G = h \cdot A^2$).

The output signal of a channel using the amplitude modulator to modulate the offset is the following:

$$Output(t) = A \cdot \cos(2\pi f_c t + \phi_c) + G \cdot AWG(t) \tag{5}$$

Options

Function	modulationType (const & value)		Description
No Modulation	AOU_MOD_OFF	0 (default)	Modulation is disabled. The channel amplitude and offset are only set by the main registers
Amplitude Modulation	AOU_MOD_AM	1	The modulating signal is used to modulate the channel amplitude
Offset Modulation	AOU_MOD_OFFSET	2	The modulating signal is used to modulate the channel offset

Table 23: Amplitude modulation options (function *modulationAmplitudeConfig*, Section 3.2.3.9 on page 47)

Programming Information

Function	Comments	Details
modulationAmplitudeConfig	It configures the amplitude modulator	Section 3.2.3.9 on page 47
AWG functions	They control the modulating signal	Section 1.2.2 on page 16
FG functions	They control the carrier signal	Section 1.2.1 on page 14

Table 24: Programming functions related to the amplitude modulator

1.4 Clock System

The SD AWG-H3300/H3300F Series uses an internally generated high-quality clock (CLKref) which is phase-locked to the chassis clock. Therefore, this clock is an extremely jitter-cleaned copy of the chassis clock. This implementation achieves a jitter and phase noise above 100 Hz which is independent of the chassis clock, depending on it only for the frequency absolute precision and long term stability. A copy of CLKref is available at the CLK connector (Table 25).

CLKref is used as a reference to generate CLKsys, the high-frequency clock used to sample data.

ADVANCED
Chassis Clock Replacement for High-Precision Applications: For applications where clock stability and precision is crucial (e.g. GPS, experimental physics, etc.), the user can replace the chassis clock with an external reference. In the case of PXI/PXIe, this is possible via a chassis clock input connector or with a PXI/PXIe timing module. These options are not available in all chassis, please check the corresponding chassis specifications.

CLK Output Options

		Programming Definitions	
Option	**Description**	**Name**	**Value**
Disable	The CLK connector is disable	n/a	0 (default)
CLKref Output	A copy of the reference clock is available at the CLK connector	n/a	1

Table 25: SD AWG-H3300/H3300F Series output clock configuration (function *clockIOconfig*, Section 3.2.3.13 on page 51)

1.4.1 FlexCLK Technology (models with variable sampling rate)

The sampling frequency of the SD AWG-H3300/H3300F Series (CLKsys frequency) can be changed using the advanced clocking system shown in Figure 24.

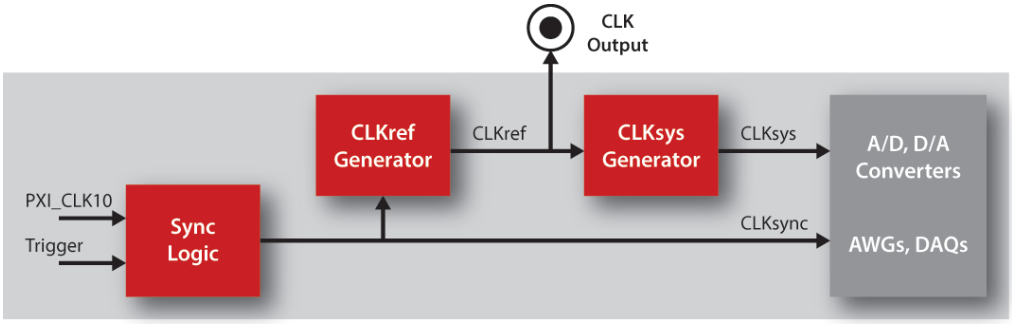


Figure 24: Functional diagram of the SD AWG-H3300/H3300F Series FlexCLK system

where:

- **CLKref** is the internal reference clock, which is phase-locked to the chassis clock.
- **CLKsys** is the system clock, used to sample data.
- **CLKsync** is an internal clock used for the synchronization features of the SD AWG-H3300/H3300F Series.
- **PXI_CLK10** is the 10 MHz clock of the PXI/PXIe backplane (PXI/PXIe only).

The CLKsys frequency can be changed within the range indicated in Table 10 on page 6 (function *clockSetFrequency*, Section 3.2.3.14 on page 52). The CLKsync frequency changes with the CLKsys frequency as follows:

$$f_{CLKsync} = GreatestCommonDivisor\left(f_{PXI\_CLK10}, \frac{f_{CLKsys}}{5}\right) \quad (6)$$

The CLKsync frequency is returned by the function *clockSetFrequency*, Section 3.2.3.14 on page 52.

CLK Set Frequency Mode

Option	Description	Programming Definitions	
		Name	Value
Low Jitter Mode	The clock system is set to achieve the lowest jitter, sacrificing tuning speed	CLK_LOW_JITTER	0
Fast Tuning Mode	The clock system is set to achieve the lowest tuning time, sacrificing jitter performance	CLK_FAST_TUNE	1

Table 26: Clock set frequency mode (function *clockSetFrequency*, Section 3.2.3.14 on page 52)

1.5 Trigger I/O

The SD AWG-H3300/H3300F Series has a general purpose input/output trigger (TRG connector). This signal can be used as general purpose digital IO or as a trigger input (Table 27), and can be sampled using the options shown in Table 28.

Trigger I/O Options

Option	Description	Programming Definitions	
		Name	Value
Trigger Output (readable)	TRG operates as a general purpose digital output signal, which can be written by the user software	AOU_TRG_OUT	0
Trigger Input	TRG operates as a trigger input, or as general purpose digital input signal, which can be read by the user software	AOU_TRG_IN	1

Table 27: SD AWG-H3300/H3300F Series TRG connector configuration (function *triggerIOconfig*, Section 3.2.3.10 on page 48)

Trigger Synchronization/Sampling Options

Option	Description	Programming Definitions	
		Name	Value
Non-synchronized mode	The trigger is sampled with an internal 100 MHz clock	SYNC_NONE	0
Synchronized mode	(PXI form factor only) The trigger is sampled using CLK10	SYNC_CLK10	1

Table 28: SD AWG-H3300/H3300F Series TRG connector configuration ((function *triggerIOconfig*, Section 3.2.3.10 on page 48)

2 Signadyne Technology and Software Overview

2.1 Programming Tools

The diagram shown in Figure 25 summarizes the programming tools available to control any Signadyne Hardware.

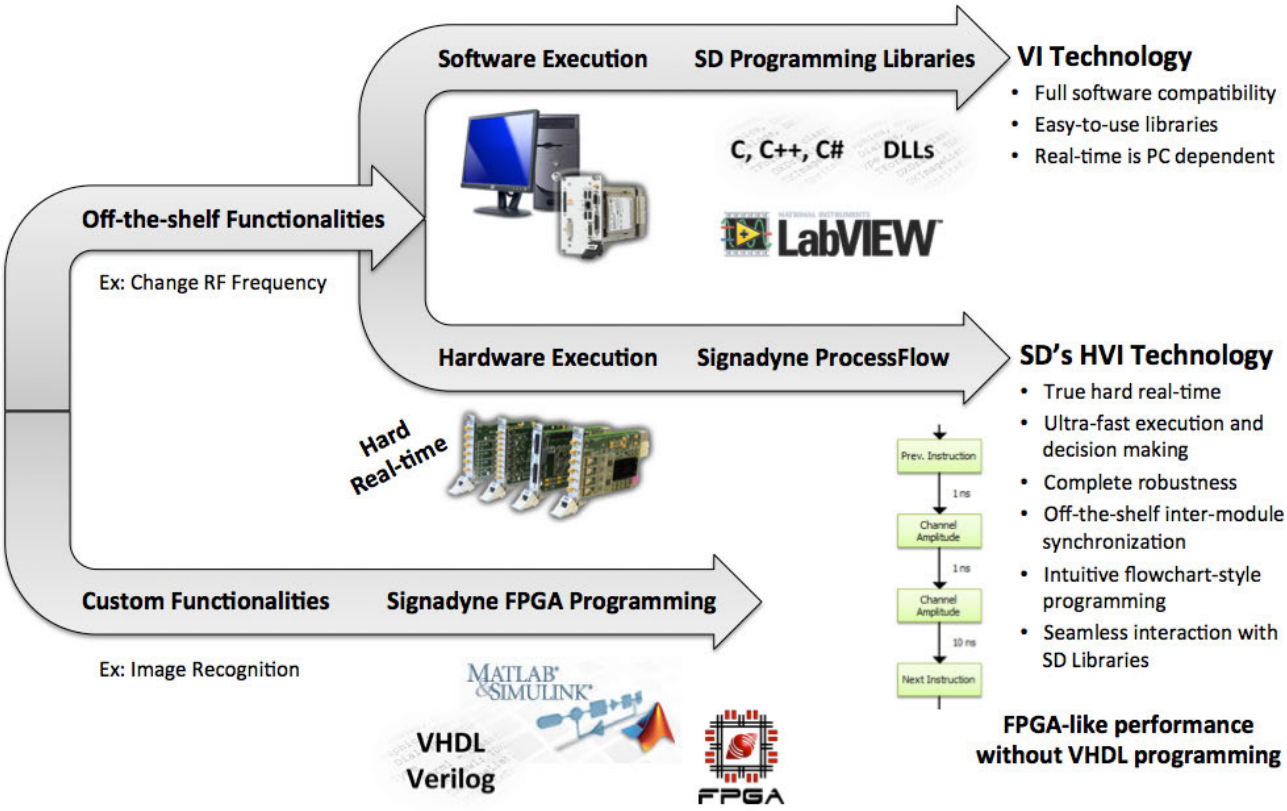


Figure 25: Programming tools for all Signadyne hardware

The diagram starts by differentiating between two main concepts:

- **Off-the-shelf Functionalities:** It refers to the control of the built-in functionalities present in all Signadyne hardware.
  For example, a built-in functionality of a Signadyne signal generator is RF generation. This functionality is managed by simple instructions that allow the user to change the RF frequency, amplitude, etc.
  Signadyne hardware is very comprehensive in terms of built-in functionalities, helping engineers to reduce development time and costs.
- **Custom Functionalities:** Some applications require the use of custom onboard real-time processing which might not be covered by the comprehensive off-the-shelf functionalities of the standard hardware products. For these applications, Signadyne provides the ”F models”, hardware products that provide the capability to program the onboard FPGA.
  For example, using the ”F model” of a Signadyne digitizer, the user has all the off-the-shelf functionalities of the hardware (data capture, triggering, etc.), but custom real-time FPGA processing can be added between the acquisition and the transmission of data to the computer.

2.1.1 Off-the-shelf Functionalities: VIs and HVIs

A comprehensive set of programming instructions controls the off-the-shelf functionalities of any Signadyne hardware. Depending on the real-time requirements of the application (timing requirements), these instructions can be executed by software, i.e. by the computer that controls the Signadyne hardware, or by the Signadyne hardware itself, thanks to an exclusive and unique technology. The first case is commonly referred as Virtual Instrumentation (VI technology), and the second case refers to Signadyne's exclusive Hard Virtual Instrumentation (HVI Technology).

2.1.1.1 Software Execution: VI Technology

Signadyne provides native programming libraries for C, C++, Visual Studio (VC++, C#, VB), MATLAB, and National Instruments LabVIEW, ensuring full software compatibility and seamless multivendor integration. Signadyne provides also dynamic libraries, e.g. DLLs, which can be used in virtually any programming language.

Figure 26: Signadyne native programming libraries ensure full compatibility, providing effortless and seamless software integration

NOTE
Virtual Instruments (VIs): The use of customizable software to create user-defined control, test and measurement systems is commonly referred as Virtual Instrumentation. In all Signadyne documentation, the concept of a Virtual Instrument (or VI) describes user software that uses programming libraries and is executed by a computer.

2.1.1.2 Hardware Execution: HVI Technology

According to the definitions described in the previous section, a VI is a piece of user-defined software executed by a computer, and therefore its real-time performance (speed, latency, etc.) is limited by the computer and by its operating system. In many cases, this real-time performance might not be enough for the application, even with a real-time operating system. In addition, many modern applications require tight triggering and precise intermodule synchronization, making the development of final systems very complex and time consuming.

For all these applications, Signadyne has developed an exclusive technology called Hard Virtual Instrumentation. In a Hard Virtual instrument (or HVI), the user program is executed by the hardware modules independently of the computer, which stays free for other VI tasks, like visualization, user interaction, etc. The I/O modules run in parallel, completely synchronized, and exchange data and decisions in real-time. The result is a set of modules that behave like a single integrated real-time instrument.

HVIs are programmed with Signadyne ProcessFlow [1], an HVI programming environment with a user-friendly flowchart-style interface, compatible with all Signadyne hardware modules.

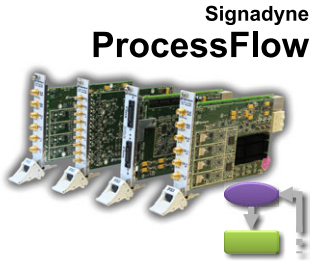


Figure 27: Signadyne ProcessFlow, a user-friendly flowchart-style HVI programming environment

NOTE
New hardware functionalities without FPGA programming: Signadyne's HVI technology is the perfect tool to create new hardware functionalities with FPGA-like performance and without any FPGA programming knowledge. Users can create a repository of HVIs that can be launched from VIs using the Signadyne Programming Libraries.

Signadyne's Hard Virtual Instrumentation technology provides:

- **Ultra-fast hard real time execution, processing and decision making:** Execution is hardware timed and can be as fast as 1 nanosecond, matching very high-performance FPGA-based systems and outperforming any real-time operating system.
- **User-friendly flowchart-style programming interface:** Signadyne ProcessFlow provides an intuitive flowchart-style programming environment that makes HVI programming extremely fast and easy (Figure 28). Using ProcessFlow and its set of built-in instructions (the same instructions available for VIs), the user can program the hardware modules without any knowledge in FPGA technology, VHDL, etc.
- **Off-the-shelf intermodule synchronization and data exchange:** Each HVI is defined by a group of hardware modules which work perfectly synchronized, without the need of any external trigger or additional external hardware (Figure 29 on the next page). In addition, Signadyne modules exchange data and decisions for ultra-fast control algorithms.
- **Complete robustness:** Execution is performed by hardware, without operating system, and independently of the user PC.
- **Seamless integration with custom FPGA functions (F models only):** HVIs can interact with user-defined FPGA functions, making the real-time processing capabilities of HVIs unlimited.
- **Seamless integration with Signadyne Programming Libraries:** In a complex control or test system, there are still some non-time-critical tasks that can only be performed by a VI, like for example: user interaction, visualization, or processing and decisions tasks which are too complex to be implemented by hardware. Therefore, in a real application, the combination of VIs and HVIs is required. This task can be performed seamlessly with the Signadyne programming tools, e.g. the user can have many HVIs and can control them from a VI using instructions like start, stop, pause, etc.

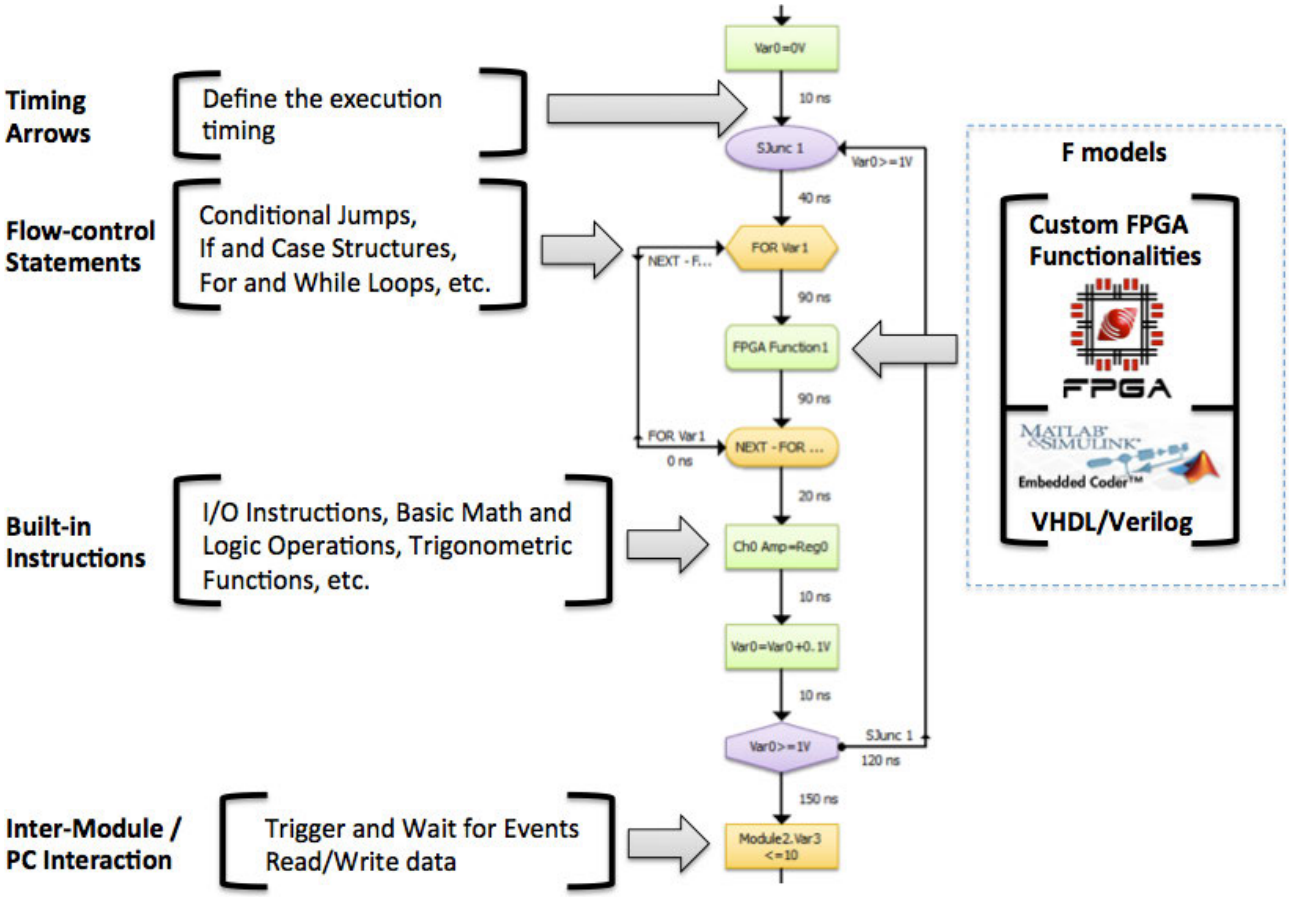


Figure 28: HVI flowchart elements. Signadyne ProcessFlow is based on flowchart programming, providing an easy-to-use environment to develop hard real-time applications

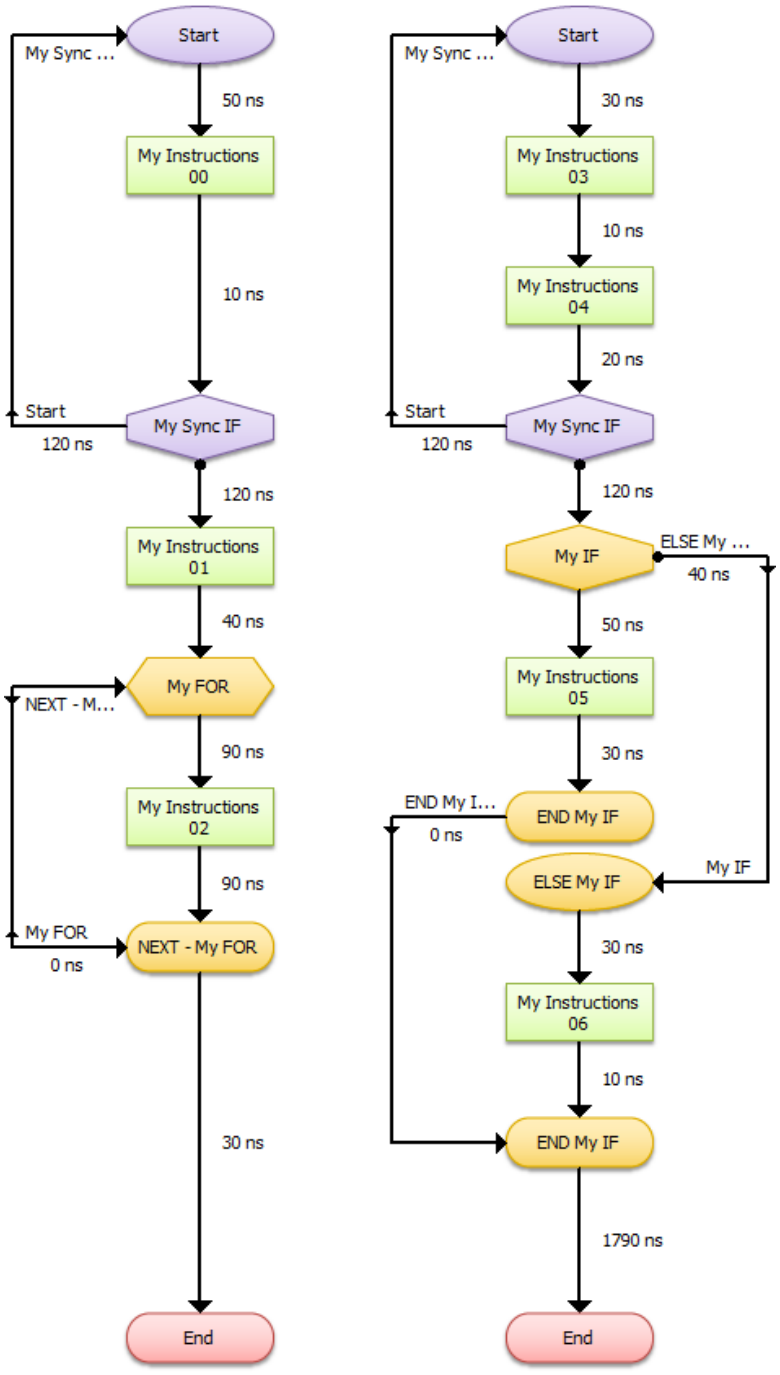


Figure 29: HVI example with two hardware modules. In an HVI, all Signadyne modules run in parallel and completely synchronized, executing one flowchart per module. This results in simpler systems without the need of triggers.

2.1.2 Custom Functionalities: FPGA Programming (F models only)

All Signadyne hardware products have an ”F” counterpart, which allows the user to program the onboard FPGA using VHDL, Verilog, or MATLAB/Simulink. The latter provides an intuitive graphical programming environment with the signal processing power and simulation capabilities of MATLAB/Simulink.

Figure 30: Signadyne FPGA technology is compatible with the most common FPGA programming languages

In addition to the FPGA programming capabilities, ”F models” provide the same built-in functionalities of their standard counterparts, giving the users more time to focus on their specific functionalities. Using a user-friendly graphical interface (Figure 31), the custom FPGA functions can be integrated into the existing firmware of the module. The user may decide to use the off-the-shelf functionalities of the hardware, or to eliminate them, releasing resources of the FPGA.

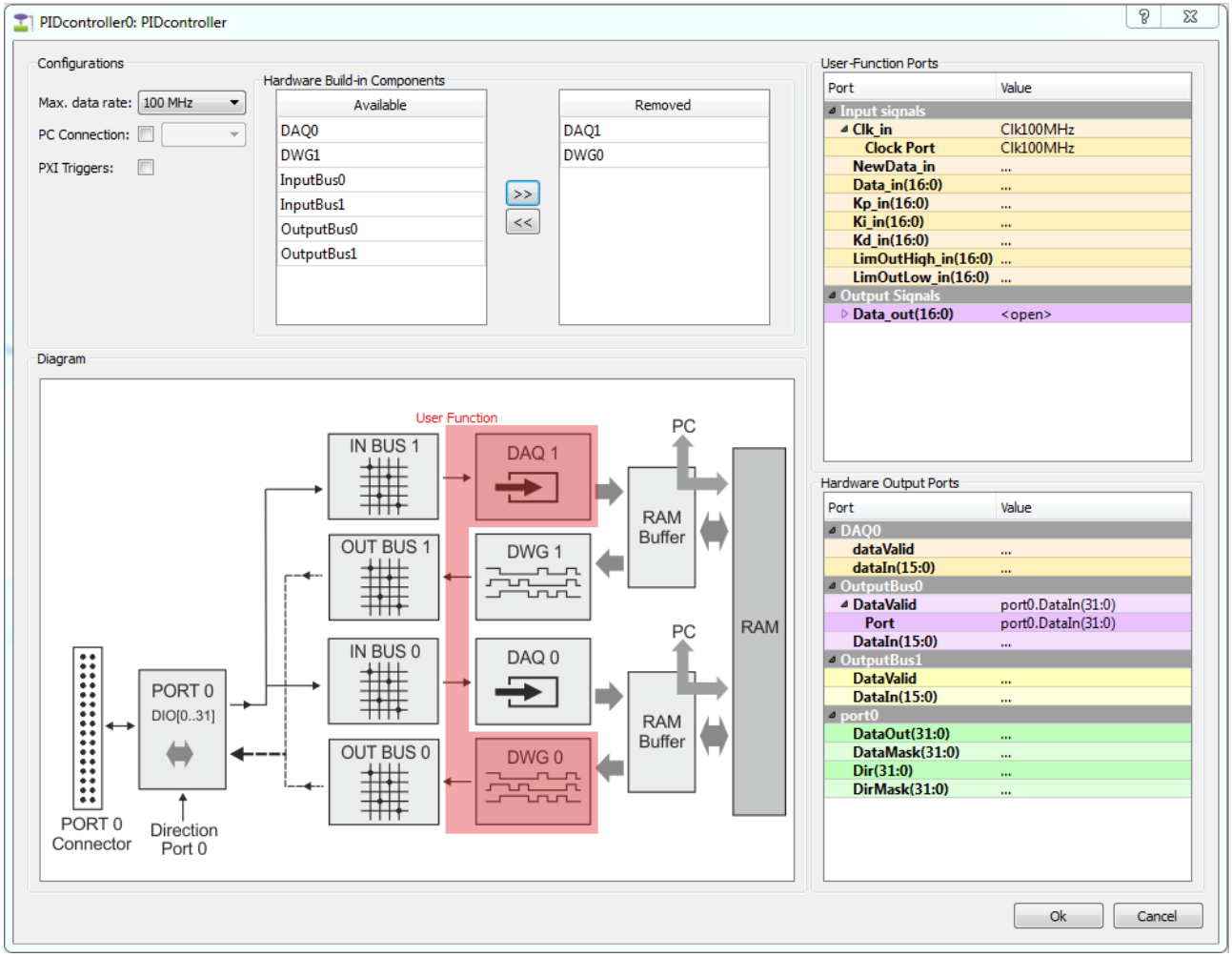


Figure 31: A Signadyne FPGA graphical IDE provides a very intuitive way of integrating the user functions with the existing hardware functionalities

NOTE
FPGA programming made simple: Full language compatibility (including the graphical environment MATLAB/Simulink) and an easy-to-use FPGA graphical IDE, make Signadyne FPGA programming extremely simple.

2.2 Application Software

2.2.1 Signadyne VirtualKnob

All Signadyne modules can be used as classical workbench instruments using Signadyne VirtualKnob [2], a ready-to-use software front panels for live operation. When VirtualKnob opens, it identifies all Signadyne hardware connected to the computer, opening the corresponding front panels.

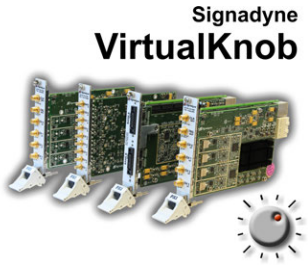


Figure 32: Signadyne VirtualKnob provides a fast and intuitive way of controlling Signadyne hardware without any programming

3 Software Tools: SD AWG-H3300/H3300F Series

3.1 VirtualKnob Front Panel

This Chapter describes the basic operations of the SD AWG-H3300/H3300F Series VirtualKnob front panel.

3.1.1 Front Panel Overview

Figure 33 shows the SD AWG-H3300/H3300F Series VirtualKnob front panel, which appears automatically when VirtualKnob is launched and the module is connected to the chassis. If there are no modules available, VirtualKnob will launch ”Demo Offline” modules.

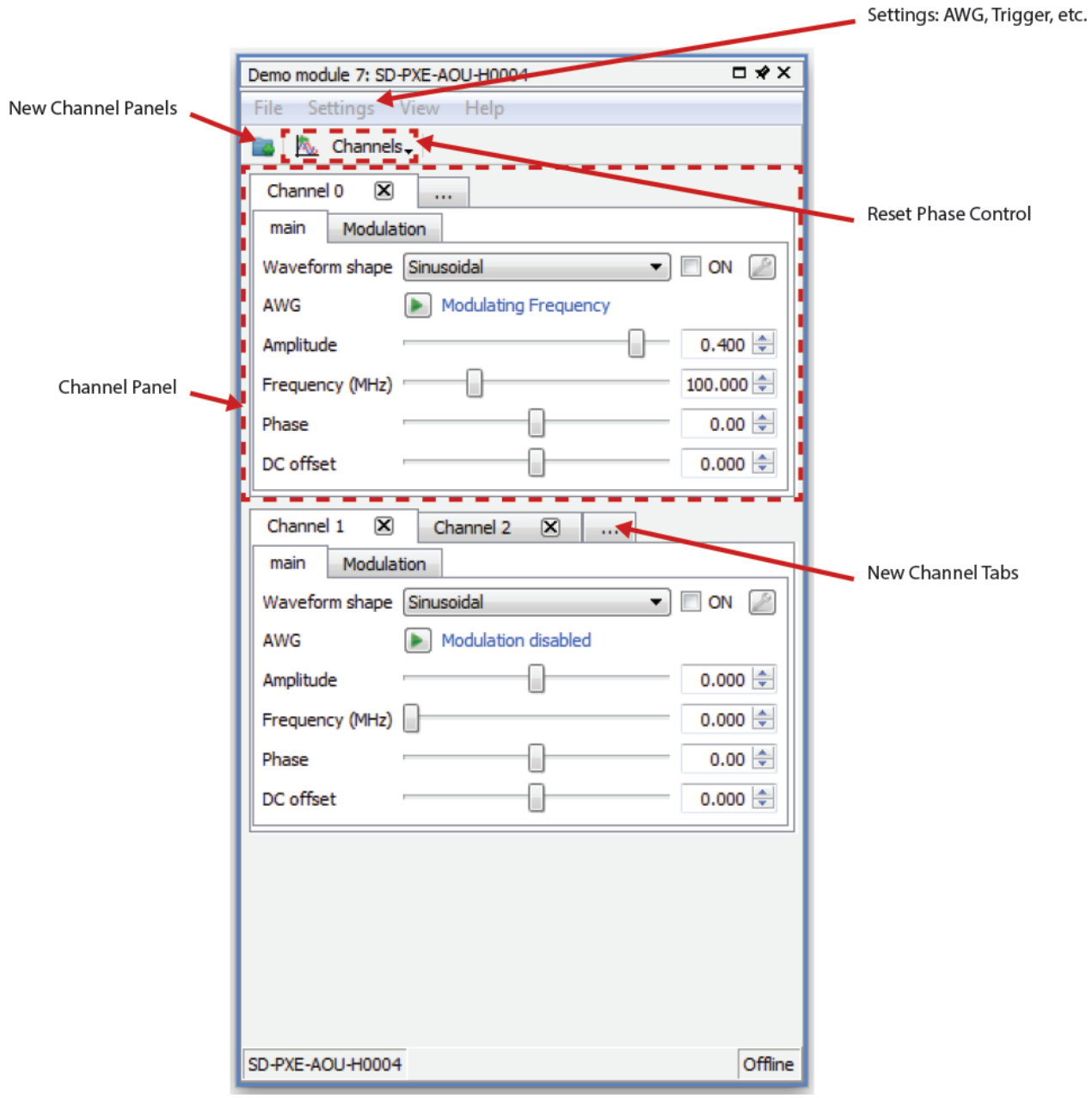


Figure 33: SD AWG-H3300/H3300F Series front panel overview.

When VirtualKnob is launched, the SD AWG-H3300/H3300F Series front panel appears empty, waiting for the user to add ”Channels”, which are windows that control the channel operation. For maximum visualization flexibility, output channels can be added as new Panels or as Tabs within an existing Panel.

3.1.2 Signal Generation

Figure 34 on the following page shows the signal generation controls of the SD AWG-H3300/H3300F Series in VirtualKnob.

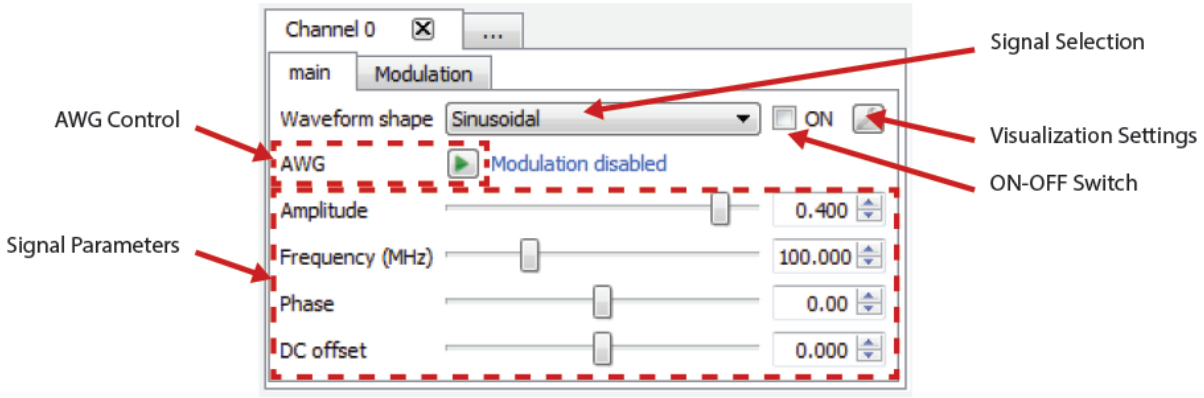


Figure 34: SD AWG-H3300/H3300F Series signal generation control.

3.1.3 Arbitrary Waveform Generation

Figure 35 shows the dialogs and the workflow to generate arbitrary waveforms in VirtualKnob using the AWGs (Section 1.2.2 on page 16).

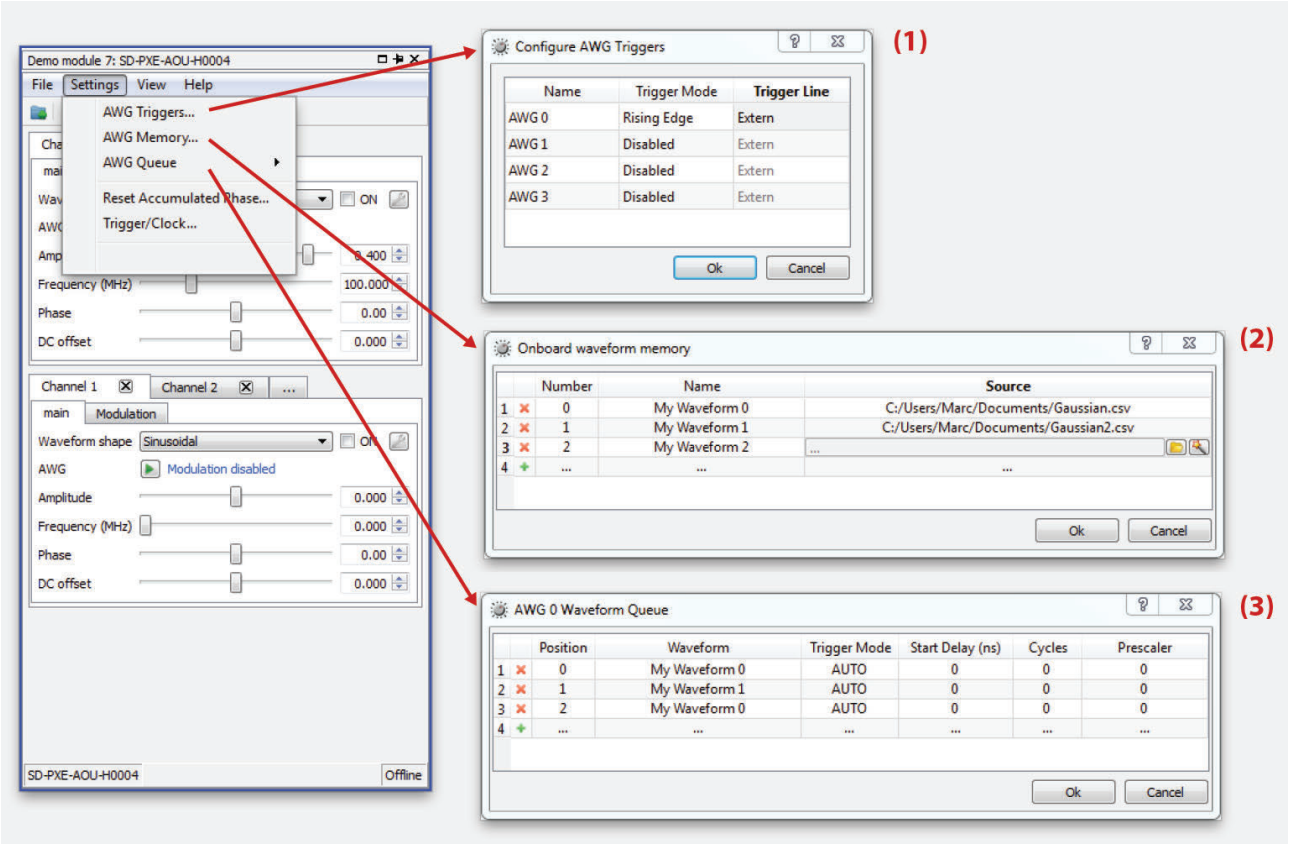


Figure 35: SD AWG-H3300/H3300F Series AWG workflow.

AWG Workflow:

1) **Configuration of the external AWG triggers (optional):** If external triggers are required, this dialog must be used to configure the source (external trigger connector, PXI trigger lines, etc.,) and the behaviour (logic level high/low, rising/falling edge, etc.).
2) **Waveforms loading:** Waveforms must be transferred to the onboard RAM in order to put them in the AWG queue. This dialog allows the user to select the waveform files and to assign custom waveform names. The waveforms are transferred to the onboard RAM when the dialog is closed.
3) **AWG queue configuration:** The waveforms loaded into the onboard RAM can be pushed into the corresponding AWG queue. This dialog allows the user to select the AWG queue options described in Section 1.2.2 on page 16.

Once the waveforms are queued in the corresponding AWG, they can be launched using the AWG control button shown in Figure 34.

3.1.4 Signal Modulation

Figure 36 shows the tab to configure the modulations in the Channel Panel.

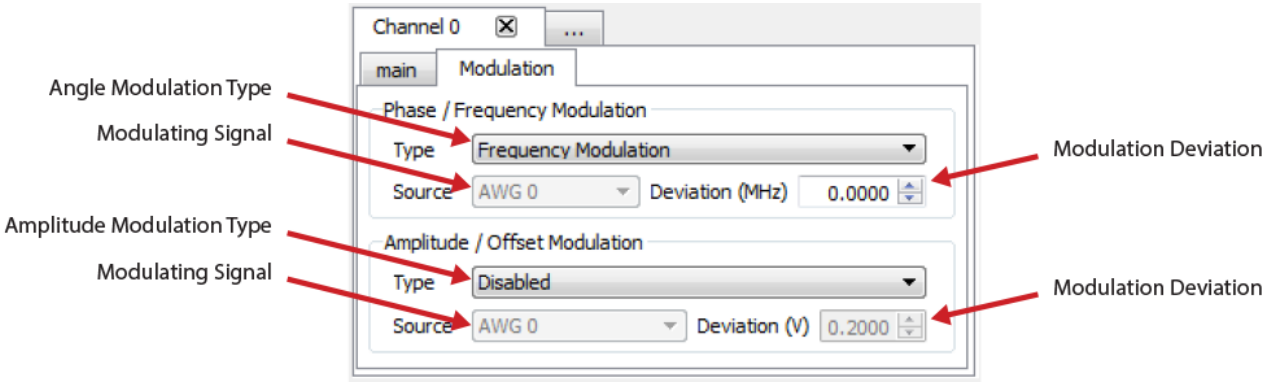


Figure 36: SD AWG-H3300/H3300F Series signal modulation control.

As described in Section 1.3 on page 24, the modulating signal for the modulators of one channel comes from the AWG corresponding to that particular channel. Therefore, the user must configure the AWG (See AWG workflow in VirtualKnob (Section 3.1.3 on the preceding page) and the modulators as shown in Figure 36. After this process, the user must run the AWG in the channel panel (Figure 33 on page 35).

3.2 Programming Functions

3.2.1 VI Programming Overview

As described in Section 2.1.1.1 on page 30, virtual instrumentation is the use of customizable software to create user-defined control, test and measurement systems, called Virtual Instruments (VI). For this purpose, Signadyne delivers highly optimized static and dynamic programming libraries [3]:

a) Static Libraries

Ready-to-use static libraries are supplied for the following programming languages and compilers:

Language	Compiler	Files
C	Any C compiler	*.h, *.lib
	MinGW (Qt), GCC	*.h, *.a
C++	MinGW (Qt), GCC	*.h, *.a
VC++, C#, VB	Microsoft Visual Studio .NET	*.dll
LabVIEW	National Instruments LabVIEW	*.vi
MATLAB		*.dll

b) Dynamic Libraries

Dynamic libraries are compatible with any programming language that has a compiler capable of performing dynamic linking. Here are some examples:

- C++ compilers not listed above.
- Other programming languages: Java, Python, PHP, Perl, Fortran, Pascal, etc.
- Computer Algebra Systems (CAS): MathWorks MATLAB, Wolfram Mathematica, Maplesoft Maple, etc.

Dynamic libraries available:

Exported Functions Language	Operating System	Files
C	Microsoft Windows	*.dll

NOTE
DLL function prototypes: The exported functions of the dynamic libraries have the same prototype as their counterparts of the static libraries

NOTE
Function Parameters: Some of the parameters of the library functions are language dependent. The table of inputs and outputs parameters for each function is a conceptual description, therefore, the user must check the specific language function to see how to use it. One example are the ID parameters (moduleID, etc.), which identify objects in non object-oriented languages. In object-oriented languages the objects are identified by their instances, and therefore the IDs are not present.
Function Names: Some programming languages like C++ or LabVIEW have a feature called function overloading or polymorphism, that allows creating several functions with the same name but with different input/output parameters. In languages without this feature, functions with different parameters must have different names.

3.2.2 HVI Programming Overview

As described in Section 2.1.1.2 on page 30, with Signadyne's exclusive hard virtual instrumentation technology the user can write custom software directly into any Signadyne hardware module, extending the concept and capabilities of virtual instrumentation. Most of the VI instructions described in this Chapter are available for HVI real-time hardware execution (programmed using Signadyne ProcessFlow [1], the HVI programming environment).

The user can find the ProcessFlow programming details and the real-time specifications of each instruction within the corresponding function description.

3.2.3 SD-AOU Class Functions

3.2.3.1 channelWaveShape

This function sets the channel output waveform type (Section 1.2 on page 13).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
waveShape	Waveshape type (Table 12 on page 13)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

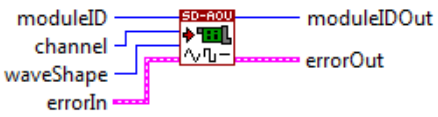
```
int SD_AOU_channelWaveShape(int moduleID, int nChannel, int waveShape);
```

C++

```
int SD_AOU::channelWaveShape(int nChannel, int waveShape);
```

LabVIEW

channelWaveShape.vi



ProcessFlow

Available: Yes

3.2.3.2 channelFrequency

This function sets the frequency for the periodic signals generated by the Function Generators (Section 1.2.1 on page 14).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
frequency	Frequency in Hz (ProcessFlow only) If the frequency is set with a register, the value must be $0..(2^{32}-1)$, where $(2^{32}-1)$ corresponds to 500 MHz
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

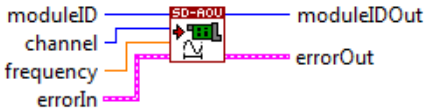
```
int SD_AOU_channelFrequency(int moduleID, int nChannel, double frequency);
```

C++

```
int SD_AOU::channelFrequency(int nChannel, double frequency);
```

LabVIEW

channelFrequency.vi



ProcessFlow

Available: Yes

3.2.3.3 channelPhase

This function sets the phase for the periodic signals generated by the Function Generators (Section 1.2.1 on page 14).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
phase	Phase in degrees (ProcessFlow only) If the phase is set with a register, the value must be $0..(2^{32}-1)$, where $(2^{32}-1)$ corresponds to 360°
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

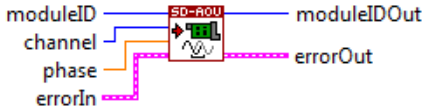
```
int SD_AOU_channelPhase(int moduleID, int nChannel, double phase);
```

C++

```
int SD_AOU::channelPhase(int nChannel, double phase);
```

LabVIEW

channelPhase.vi



ProcessFlow

Available: Yes

3.2.3.4 channelPhaseReset

This function resets the accumulated phase of the selected channel. This accumulated phase is the result of the phase continuous operation of the product (Section 1.2.1 on page 14).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Channel to reset
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_channelPhaseReset(int moduleID, int nChannel);
```

C++

```
int SD_AOU::channelPhaseReset(int nChannel);
```

LabVIEW

channelPhaseReset.vi

ProcessFlow

Available: Yes

3.2.3.5 channelPhaseResetMultiple

This function resets the accumulated phase of the selected channels simultaneously. This accumulated phase is the result of the phase continuous operation of the product (Section 1.2.1 on page 14).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
channelMask	Mask to select the channels to reset (LSB is channel 0, bit 1 is channel 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_channelPhaseResetMultiple(int moduleID, int channelMask);
```

C++

```
int SD_AOU::channelPhaseResetMultiple(int channelMask);
```

LabVIEW

channelPhaseResetMultiple.vi

ProcessFlow

Available: No

3.2.3.6 channelAmplitude

This function sets the amplitude of a channel (Section 1.2.3 on page 23).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
amplitude	Amplitude in volts (ProcessFlow only) If the amplitude is set with a register, the value must be $-(2^{(bits-1)})..(2^{(bits-1)}-1)$, where $bits$ corresponds to the amplitude resolution bits (see Specifications on page 3)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

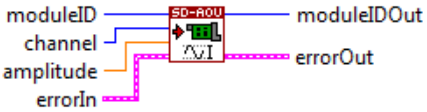
```
int SD_AOU_channelAmplitude(int moduleID, int nChannel, double amplitude);
```

C++

```
int SD_AOU::channelAmplitude(int nChannel, double amplitude);
```

LabVIEW

channelAmplitude.vi



ProcessFlow

Available: Yes

3.2.3.7 channelOffset

This function sets the DC offset of a channel (Section 1.2.3 on page 23).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
offset	DC offset in volts (ProcessFlow only) If the DC offset is set with a register, the value must be $-(2^{(bits-1)})..(2^{(bits-1)}-1)$, where $bits$ corresponds to the amplitude resolution bits (see Specifications on page 3)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

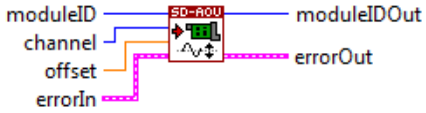
```
int SD_AOU_channelOffset(int moduleID, int nChannel, double offset);
```

C++

```
int SD_AOU::channelOffset(int nChannel, double offset);
```

LabVIEW

channelOffset.vi



ProcessFlow

Available: Yes

3.2.3.8 modulationAngleConfig

This function configures the modulation in frequency/phase for the selected channel (Section 1.3.1 on page 24).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
modulationType	Modulation type (Table 21 on page 24)
deviationGain	Gain for the modulating signal
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

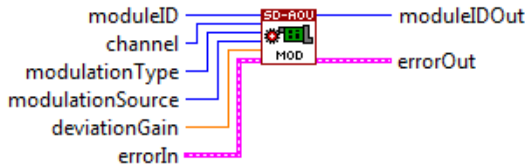
int SD_AOU_modulationAngleConfig(int moduleID, int nChannel, int modulationType, int deviationGain);

C++

int SD_AOU::modulationAngleConfig(int nChannel, int modulationType, int deviationGain);

LabVIEW

modulationAngleConfig.vi



ProcessFlow

Available: Yes

3.2.3.9 modulationAmplitudeConfig

This function configures the modulation in amplitude/offset for the selected channel (Section 1.3.2 on page 26).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nChannel	Output channel number
modulationType	Modulation type (Table 23 on page 26)
deviationGain	Gain for the modulating signal
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

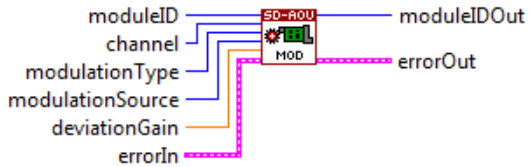
int SD_AOU_modulationAmplitudeConfig(int moduleID, int nChannel, int modulationType, int deviationGain);

C++

int SD_AOU::modulationAmplitudeConfig(int nChannel, int modulationType, int deviationGain);

LabVIEW

modulationAmplitudeConfig.vi



ProcessFlow

Available: Yes

3.2.3.10 triggerIOconfig

This function configures the trigger connector direction and synchronization/sampling method (TRG, Section 1.5 on page 28).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
direction	Input or output (Table 27 on page 28)
syncMode	Sampling/synchronization mode (Table 28 on page 28)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

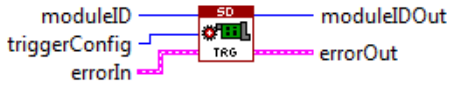
```
int SD_AOU_triggerIOconfig(int moduleID, int direction, int syncMode);
```

C++

```
int SD_AOU::triggerIOconfig(int direction, int syncMode);
```

LabVIEW

triggerIOconfig.vi



ProcessFlow

Available: No

3.2.3.11 triggerIOwrite

This function sets the trigger output. The trigger must be configured as output using function *triggerIOconfig*, Section 3.2.3.10 on the preceding page (see Trigger Section 1.5 on page 28).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
value	Trigger output value: 0 (OFF), 1 (ON)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

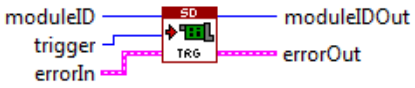
```
int SD_AOU_triggerIOwrite(int moduleID, int value);
```

C++

```
int SD_AOU::triggerIOwrite(int value);
```

LabVIEW

triggerIOwrite.vi



ProcessFlow

Available: Yes

3.2.3.12 triggerIOread

This function reads the trigger input (Section 1.5 on page 28)

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
value	Trigger input value: 0 (OFF), 1 (ON). Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

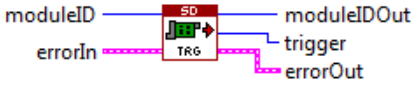
```
int SD_AOU_triggerIOread(int moduleID);
```

C++

```
int SD_AOU::triggerIOread();
```

LabVIEW

triggerIOread.vi



ProcessFlow

Available: No

3.2.3.13 clockIOconfig

This function configures the operation of the clock output connector (CLK, Section 1.4 on page 27).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
clockConfig	Clock connector function (Table 25 on page 27)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

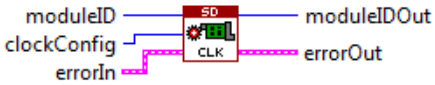
```
int SD_AOU_clockIOconfig(int moduleID, int clockConfig);
```

C++

```
int SD_AOU::clockIOconfig(int clockConfig);
```

LabVIEW

clockIOconfig.vi



ProcessFlow

Available: No

3.2.3.14 clockSetFrequency

This function sets the module clock frequency (CLKsys, Section 1.4.1 on page 27).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
frequency	Frequency in Hz (Table 10 on page 6)
mode	Operation mode of the variable clock system (Table 26 on page 28)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
CLKsysFreq*	It returns the real frequency applied to the hardware in Hz. It may differ from the desired frequency due to the hardware frequency resolution. Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

*In Signadyne Programming Libraries v.1.57.61 or older, clockSetFrequency returns CLKsyncFreq, the frequency of the internal CLKsync in Hz (Equation 6 on page 27)

C

```
double SD_AOU_clockSetFrequency(int moduleID, double frequency, int mode);
```

C++

```
double SD_AOU::clockSetFrequency(double frequency, int mode);
```

LabVIEW

clockSetFrequency.vi

ProcessFlow

Available: No

3.2.3.15 clockGetFrequency

This function returns the real hardware clock frequency (CLKsys, Section 1.4.1 on page 27). It may differ from the frequency set with the function *clockSetFrequency*, Section 3.2.3.14 on the preceding page, due to the hardware frequency resolution.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
CLKsysFreq	It returns the real hardware clock frequency in Hz (CLKsys). Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

```
double SD_AOU_clockGetFrequency(int moduleID);
```

C++

```
double SD_AOU::clockGetFrequency();
```

LabVIEW

clockGetFrequency.vi

ProcessFlow

Available: No

3.2.3.16 clockGetSyncFrequency

This function returns the frequency of CLKsync (Section 1.4.1 on page 27).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
CLKsyncFreq	It returns the frequency of the internal CLKsync in Hz (Equation 6 on page 27). Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

```
int SD_AOU_clockGetSyncFrequency(int moduleID);
```

C++

```
int SD_AOU::clockGetSyncFrequency();
```

LabVIEW

clockGetSyncFrequency.vi

ProcessFlow

Available: No

3.2.3.17 clockResetPhase

This function sets the module in a sync state, waiting for the first trigger to reset the phase of the internal clocks CLKsync and CLKsys (Section 1.4.1 on page 27, and Section 1.2.2.6 on page 19).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
triggerBehavior	Trigger behaviour (Table 17 on page 19)
PXItrigger	PXI trigger number as shown in Table 16 on page 19
skew	Skew between PXI_CLK10 and CLKsync in multiples of 10 ns
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDout	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_clockResetPhase(int moduleID, int triggerBehavior, int PXItrigger, double skew);
```

C++

```
int SD_AOU::clockResetPhase(int triggerBehavior, int PXItrigger, double skew);
```

LabVIEW

clockResetPhase.vi

ProcessFlow

Available: No

3.2.3.18 waveformLoad

This function loads the specified waveform into the module onboard RAM. Waveforms must be created first with the SD-Wave class (Section 3.2.5 on page 90).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
waveformID	Waveform identifier (see SD-Wave Class, Section 3.2.5 on page 90)
waveformObject	Pointer to the waveform object (see SD-Wave Class, Section 3.2.5 on page 90)
waveformType	Waveform Type (Table 18 on page 21)
waveformPoints	Number of points of the waveform, which must be a multiple of a certain number of points (according to the AWG specifications)
waveformDataRaw	Array with waveform points. In dual and IQ waveforms, the waveform points are interleaved (WaveformA0, WaveformB0, WaveformA1, etc.)
waveformNumber	Waveform number to identify the waveform in subsequent related function calls. This value must be in the [0..n] range, and in order to optimized onboard memory usage, it should as low as possible
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
availableRAM	Available onboard RAM in waveform points, or a negative number for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C

int SD_AOU_waveformLoad(int moduleID, int waveformID, int waveformNumber);
int SD_AOU_waveformLoadArrayInt16(int moduleID, int waveformType, int waveformPoints, short* waveformDataRaw, int waveformNumber);

C++

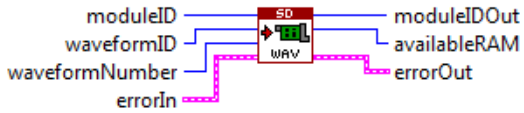
int SD_AOU::waveformLoad(SD_Wave* waveformObject, int waveformNumber);
int SD_AOU::waveformLoad(int waveformType, int waveformPoints, short* waveformDataRaw, int waveformNumber);

Visual Studio (VC++, C#, VB)

int SD_AOU::waveformLoad(SD_Wave waveformObject, int waveformNumber);
int SD_AOU::waveformLoad(int waveformType, short[] waveformDataRaw, int waveformNumber);

LabVIEW

waveformLoad.vi



3.2.3.19 waveformReLoad

This function replaces a waveform located in the module onboard RAM. The size of the new waveform must be smaller than or equal to the existing waveform.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
waveformID	Waveform identifier (see SD-Wave Class, Section 3.2.5 on page 90)
waveformObject	Pointer to the waveform object (see SD-Wave Class, Section 3.2.5 on page 90)
waveformType	Waveform Type (Table 18 on page 21)
waveformPoints	Number of points of the waveform, which must be a multiple of a certain number of points (according to the AWG specifications). If this is not the case, paddingMode can be used.
waveformDataRaw	Array with waveform points. In dual and IQ waveforms, the waveform points are interleaved (WaveformA0, WaveformB0, WaveformA1, etc.)
waveformNumber	Waveform number that identifies the waveform that must be replaced
paddingMode	if 0, the waveform is loaded as it is, and zeros are added at the end if the number of points is not a multiple of the number required by the AWG. If 1, the waveform is loaded n times (using DMA) until the total number of points is multiple of the number required by the AWG (it only works for waveforms with a even number of points).
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
availableRAM	Available onboard RAM in waveform points, or a negative number for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C

int SD_AOU_waveformReLoad(int moduleID, int waveformID, int waveformNumber, int paddingMode=0);
int SD_AOU_waveformReLoadArrayInt16(int moduleID, int waveformType, int waveformPoints, short* waveformDataRaw, int waveformNumber, int paddingMode=0);

C++

int SD_AOU::waveformReLoad(SD_Wave* waveformObject, int waveformNumber, int paddingMode=0);
int SD_AOU::waveformReLoad(int waveformType, int waveformPoints, short* waveformDataRaw, int waveformNumber, int paddingMode=0);

Visual Studio (VC++, C#, VB)

int SD_AOU::waveformReLoad(SD_Wave waveformObject, int waveformNumber, int paddingMode=0);
int SD_AOU::waveformReLoad(int waveformType, short[] waveformDataRaw, int waveformNumber, int paddingMode=0);

LabVIEW

waveformReLoad.vi

3.2.3.20 waveformFlush

This function deletes all the waveforms from the module onboard RAM and flushes all the AWG queues (*AWGflush*, Section 3.2.3.23 on page 62).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_waveformFlush(int moduleID);
```

C++

```
int SD_AOU::waveformFlush();
```

LabVIEW

waveformFlush.vi

3.2.3.21 AWG

This function provides a one-step method to load, queue and start a single waveform in one of the module Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16). The waveform can be loaded from an array of points in memory or from a file.

ADVANCED
Step-by-step programming: This function is equivalent to create a waveform with *new* (SD-Wave class, Section 3.2.5 on page 90), and to call *waveformLoad*, *AWGqueueWaveform* and *AWGstart*. By using these functions step by step, the user has complete control of the memory usage, data transfer times between the PC and the module, and the possibility to create generation sequences and to control the generation start time precisely (Section 1.2.2 on page 16).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number where the waveform is queued
triggerMode	Trigger method to launch the waveform (Table 15 on page 18)
startDelay	Defines the delay between the trigger and the waveform launch in tensof ns
cycles	Number of times the waveform is repeated once launched (negative means infinite)
prescaler	Waveform prescaler value, to reduce the effective sampling rate (Equation 1 on page 18)
waveformType	Waveform Type (Table 18 on page 21)
waveformPoints	Number of points of the waveform
waveformDataA	Array with waveform points. Analog waveforms are defined with floating point numbers, which correspond to a normalized amplitude (-1 to 1)
waveformDataB	Array with waveform points, only for the waveforms which have a second component (e.g. Q in IQ modulations defined in cartesian, or phase in IQ modulations defined with polars)
waveformFile	File containing the waveform points
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
availableRAM	Available onboard RAM in waveform points, or negative number for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C Function

int SD_AOU_AWGfromArray(int moduleID, int nAWG, int triggerMode, int startDelay, int cycles, int prescaler,
int waveformType, int waveformPoints, double* waveformDataA, double* waveformDataB=0);

int SD_AOU_AWGfromFile(int moduleID, int nAWG, char* waveformFile, int triggerMode, int startDelay, int cycles, int prescaler);

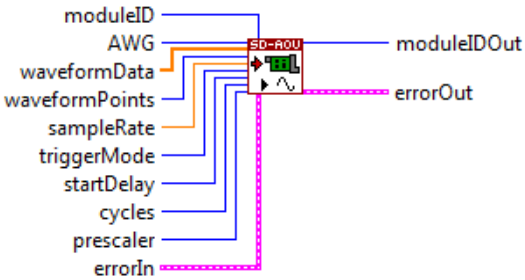
C++ Function

int SD_AOU::AWG(int nAWG, int triggerMode, int startDelay, int cycles, int prescaler, int waveformType,
int waveformPoints, double* waveformDataA, double* waveformDataB=0);

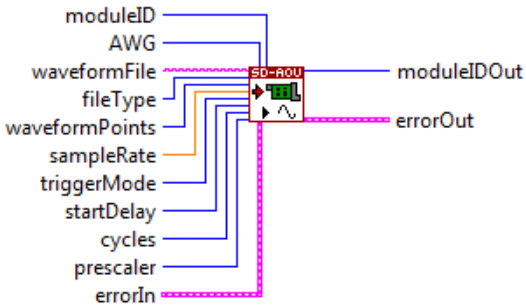
int SD_AOU::AWG(int nAWG, char* waveformFile, int triggerMode, int startDelay, int cycles, int prescaler);

LabVIEW

SD_Module_AWGfromArray.vi



SD_Module_AWGfromFile.vi



3.2.3.22 AWGqueueWaveform

This function queues the specified waveform in one of the Arbitrary Waveform Generators (AWGs) of the module (Section 1.2.2 on page 16). The waveform must be already loaded in the module onboard RAM (function *waveformLoad*, Section 3.2.3.18 on page 56).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number where the waveform is queued
waveformNumber	Waveform to be queued into the AWG. It must be already loaded with *WaveformLoad*
triggerMode	Trigger method to launch the waveform (Table 15 on page 18)
startDelay	Defines the delay between the trigger and the waveform launch in tens of ns
cycles	Number of times the waveform is repeated once launched (negative means infinite)
prescaler	Waveform prescaler value, to reduce the effective sampling rate (Equation 1 on page 18)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGqueueWaveform(int moduleID, int nAWG, int waveformNumber, int triggerMode, int startDelay, int cycles, int prescaler);
```

C++

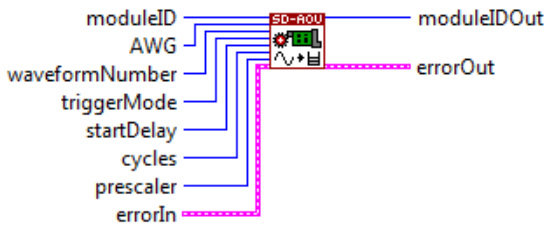
```
int SD_AOU::AWGqueueWaveform(int nAWG, int waveformNumber, int triggerMode, int startDelay, int cycles, int prescaler);
```

ProcessFlow

- AWGqueueWaveform
  - AWG
  - Waveform number
  - Trigger Mode
  - Start Delay
    - Type
    - Value
    - Cycles

LabVIEW

AWGqueueWaveform.vi



3.2.3.23 AWGflush

This function empties the queue of the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16). Waveforms are not removed from the module onboard RAM.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGflush(int moduleID, int nAWG);
```

C++

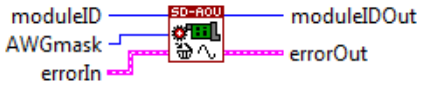
```
int SD_AOU::AWGflush(int nAWG );
```

ProcessFlow

- AWGflush
  - Channel

LabVIEW

AWGflush.vi



3.2.3.24 AWGstart

This function starts the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16) from the beginning of its queue. The generation will start immediately or when a trigger is received, depending on the trigger selection of the first waveform in the queue and provided that at least one waveform is queued in the AWG (functions *AWGqueueWaveform*, or *AWG*).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGstart(int moduleID, int nAWG);
```

C++

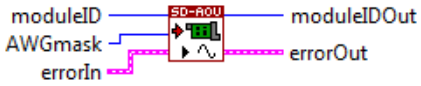
```
int SD_AOU::AWGstart(int nAWG);
```

ProcessFlow

◢ AWGstart
AWG

LabVIEW

AWGstart.vi



3.2.3.25 AWGstartMultiple

This function starts the selected Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16) from the beginning of their queues. The generation will start immediately or when a trigger is received, depending on the trigger selection of the first waveform in their queues and provided that at least one waveform is queued in these AWGs (functions *AWGqueueWaveform*, or *AWG*).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs to be started (LSB is AWG 0, bit 1 is AWG 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGstartMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGstartMultiple(int AWGmask);
```

LabVIEW

AWGstartMultiple.vi

3.2.3.26 AWGpause

This function pauses the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16), leaving the last waveform point at the output, and ignoring all incoming triggers. The waveform generation can be resumed calling *AWGresume*.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGpause(int moduleID, int nAWG);
```

C++

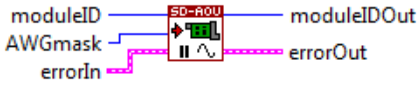
```
int SD_AOU::AWGpause(int nAWG);
```

ProcessFlow

- AWGpause
  - AWG

LabVIEW

AWGpause.vi



3.2.3.27 AWGpauseMultiple

This function pauses the selected Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16), leaving the last waveform point at the output, and ignoring all incoming triggers. The waveform generation can be resumed calling *AWGresumeMultiple*.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs to be paused (LSB is AWG 0, bit 1 is AWG 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGpauseMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGpauseMultiple(int AWGmask);
```

3.2.3.28 AWGresume

This function resumes the operation of the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16) from the current position of the queue.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGresume(int moduleID, int nAWG);
```

C++

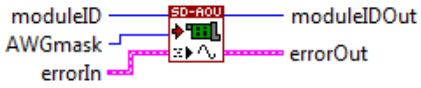
```
int SD_AOU::AWGresume(int nAWG);
```

ProcessFlow

- AWGresume
  - AWG

LabVIEW

AWGresume.vi



3.2.3.29 AWGresumeMultiple

This function resumes the operation of the selected Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16) from the current position of their respective queues.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs to be resumed (LSB is AWG 0, bit 1 is AWG 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGresumeMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGresumeMultiple(int AWGmask);
```

LabVIEW

AWGresumeMultiple.vi

3.2.3.30 AWGstop

This function stops the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16), setting the output to zero and resetting the AWG queue to its initial position. All following incoming triggers are ignored.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGstop(int moduleID, int nAWG);
```

C++

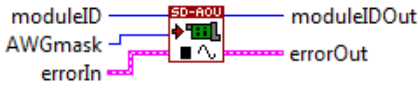
```
int SD_AOU::AWGstop(int nAWG);
```

ProcessFlow

- AWGstop
  - AWG

LabVIEW

AWGstop.vi



3.2.3.31 AWGstopMultiple

This function stops the selected Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16), setting their outputs to zero and resetting their respective queues to the initial positions. All following incoming triggers are ignored.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs to be stopped (LSB is AWG 0, bit 1 is AWG 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGstopMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGstopMultiple(int AWGmask);
```

LabVIEW

AWGstopMultiple.vi

3.2.3.32 AWGjumpNextWaveform

This function forces a jump to the next waveform in the AWG queue. The jump is executed once the current waveform has finished a complete cycle.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGjumpNextWaveform(int moduleID, int nAWG);
```

C++

```
int SD_AOU::AWGjumpNextWaveform(int nAWG);
```

ProcessFlow

- AWGjumpNextWaveform
  - AWG

LabVIEW

AWGjumpNextWaveform.vi

3.2.3.33 AWGjumpNextWaveformMultiple

This function forces a jump to the next waveform in the queue of several AWGs. The jumps are executed once the current waveforms have finished a complete cycle.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs (LSB is AWG 0, bit 1 is AWG 1 and so forth)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGjumpNextWaveformMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGjumpNextWaveformMultiple(int AWGmask);
```

LabVIEW

AWGjumpNextWaveformMultiple.vi

3.2.3.34 AWGisRunning

This function returns if the AWG is running or stopped.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
running	1 if the AWG is running, 0 if it is stopped
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGisRunning(int moduleID, int nAWG);
```

C++

```
int SD_AOU::AWGisRunning(int nAWG);
```

ProcessFlow

- AWGisRunning
  - AWG

LabVIEW

AWGisRunning.vi

3.2.3.35 AWGnWFplaying

This function returns the waveformNumber of the waveform which is currently being generated.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
waveformNumber	waveform identifier (see function *waveformLoad*, Section 3.2.3.18 on page 56)
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGnWFplaying(int moduleID, int nAWG);
```

C++

```
int SD_AOU::AWGnWFplaying(int nAWG);
```

ProcessFlow

- AWGnWFplaying
  - AWG

LabVIEW

AWGnWFplaying.vi

3.2.3.36 AWGtriggerExternalConfig

This function configures the external triggers for the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16). The external trigger is used in case the waveform is queued with the external trigger mode option (function AWGqueueWaveform, Section 3.2.3.22 on page 61).

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
externalSource	External trigger source (Table 16 on page 19)
triggerBehavior	Trigger behaviour (Table 17 on page 19)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGtriggerExternalConfig(int moduleID, int nAWG, int externalSource, int triggerBehaviour);
```

C++

```
int SD_AOU::AWGtriggerExternalConfig(int nAWG, int externalSource, int triggerBehaviour);
```

LabVIEW

AWGtriggerExternalConfig.vi

3.2.3.37 AWGtrigger

This function triggers the selected Arbitrary Waveform Generator (AWG) (Section 1.2.2 on page 16). The waveform waiting in the current position of the queue is launched provided it is configured with VI/HVI Trigger.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nAWG	AWG number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGtrigger(int moduleID, int nAWG);
```

C++

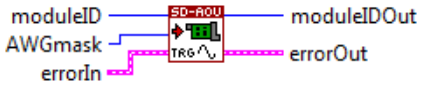
```
int SD_AOU::AWGtrigger(int nAWG);
```

ProcessFlow

- AWGtrigger
  - AWG

LabVIEW

AWGtrigger.vi



3.2.3.38 AWGtriggerMultiple

This function triggers the selected Arbitrary Waveform Generators (AWGs) (Section 1.2.2 on page 16). The waveform waiting in the current position of the queue is launched provided it is configured with VI/HVI Trigger.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
AWGmask	Mask to select the AWGs to be triggered (LSB is AWG 0, bit 1 is AWG 1 and so forth)
AWG	AWG to be triggered
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_AOU_AWGtriggerMultiple(int moduleID, int AWGmask);
```

C++

```
int SD_AOU::AWGtriggerMultiple(int AWGmask);
```

ProcessFlow

- AWGtriggerMultiple
  - AWG

LabVIEW

AWGtriggerMultiple.vi

3.2.3.39 MathAssign (D=S)

This function copies the value of the source parameter (S) into the destination variable (D).

Parameters

Name	Description
Inputs	
destination	Destination local variable
source	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
-	Source local variable
-	Digital value of the trigger input

ProcessFlow

- MathAssign (D=S)
  - Destination
  - Source
    - Type
    - Value

3.2.3.40 MathArithmetics (R=A[+-*/]B)

This function subtracts, adds, multiplies or divides the values of the operands A and B, writing the result in R.

Parameters

Name	Description
Inputs	
Result	Destination local variable
A	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
	Local variable
-	Digital value of the trigger input
Operation	Arithmetic operation (+, -, *, /)
B	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
	Local variable
-	Digital value of the trigger input

ProcessFlow

- MathArithmetics (R=A[+-*/]B)
  - Result
  - A
    - Type
    - Value
  - Operation
  - B
    - Type
    - Value

3.2.3.41 MathMultAcc (R=A[+-]B[*/]C)

This function performs a Multiplication and Accumulation (MAC) operation, widely used in digital signal processing.

Parameters

Name	Description
Inputs	
Result	Destination local variable
A	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
	Local variable
-	Digital value of the trigger input
Operation	Arithmetic operation for the Accumulation (+ or -)
B	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
	Local variable
-	Digital value of the trigger input
Operation	Arithmetic operation for the Multiplication (* or /)
C	Constant in the specified format (Decimal, Binary, Hexadecimal, etc.)
	Local variable
-	Digital value of the trigger input

ProcessFlow

- MathMultAcc (R=A[+-]B[*/]C)
  - Result
  - A
    - Type
    - Value
    - Operation
  - B
    - Type
    - Value
    - Operation
  - C
    - Type
    - Value

3.2.4 SD-Module Class Functions

3.2.4.1 open

This function initializes a hardware module, therefore it must be called before using any other module-related function. A module can be opened using the serial number or the chassis and slot number. The first option ensures the same module is always opened regardless its chassis or slot location.

Parameters

Name	Description
Inputs	
productName	Complete product name (e.g. ”SD AOU-H3334-PXIe-1G”). This name can be found on the product, in Signadyne Manager (SDM), or in nearly any Signadyne software. It can also be retrieved with the function *getProductName*, Section 3.2.4.4 on page 84
serialNumber	Module Serial Number (e.g. ”ND23G86A”). The serial number can be found on the product, in Signadyne Manager (SDM), or in nearly any Signadyne software. It can also be retrieved with the function *getSerialNumber*, Section 3.2.4.5 on page 85
chassis	Chassis number where the device is located. The chassis number can be found in Signadyne Manager (SDM) , or in nearly any Signadyne software. It can also be retrieved with the function *getChassis*, Section 3.2.4.6 on page 86
slot	Slot number where the device is plugged in. This number can be found on the chassis, in Signadyne Manager (SDM), or in nearly any Signadyne software. It can also be retrieved with the function *getSlot*, Section 3.2.4.7 on page 87
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleID	(Non-object-oriented languages only) Module identifier, or a negative number for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C Function

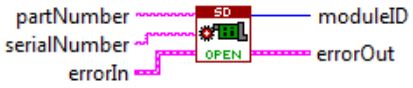
```
int SD_Module_openWithSerialNumber(char* productName, char* serialNumber);
int SD_Module_openWithSlot(char* productName, int chassis, int slot);
```

C++ Function

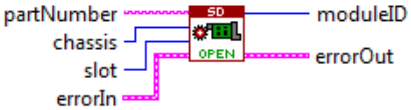
```
int SD_Module::open(char* productName, char* serialNumber);
int SD_Module::open(char* productName, int chassis, int slot);
```

LabVIEW

SD_Module_openWithSerialNumber.vi



SD_Module_openWithSlot.vi



3.2.4.2 close

This function releases all the resources allocated for the module instance. It must be always called before exiting the application.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on the previous page
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
errorOut	(LabVIEW only) A copy of moduleID

C

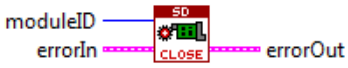
```
int SD_Module_close(int moduleID);
```

C++

```
int SD_Module::close();
```

LabVIEW

close.vi



3.2.4.3 moduleCount

This function returns the number of Signadyne modules installed in the system.

NOTE
Static Function: (Object-oriented languages only) moduleCount is a static function

Parameters

Name	Description
Inputs	
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
nModules	Number of Signadyne modules installed in the system. Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

```
int SD_Module_moduleCount();
```

C++

```
int SD_Module::moduleCount();
```

LabVIEW

moduleCount.vi

ProcessFlow

Available: No

3.2.4.4 getProductName

This function returns the product name of the specified device.

NOTE
Static Function: (Object-oriented languages only) getProductName is a static function

Parameters

Name	Description
Inputs	
index	Module index. It must be in the range 0..(nModules-1), where nModules is returned by function *moduleCount*, Section 3.2.4.3 on the preceding page
chassis	Chassis number where the device is located. The chassis number can be found in Signadyne Manager (SDM) , or in nearly any Signadyne software. It can also be retrieved with the function *getChassis*, Section 3.2.4.6 on page 86
slot	Slot number where the device is plugged in. This number can be found on the chassis, in Signadyne Manager (SDM), or in nearly any Signadyne software. It can also be retrieved with the function *getSlot*, Section 3.2.4.7 on page 87
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
productName	Product name. It can be used in function *open*, Section 3.2.4.1 on page 81
errorOut	See error codes in Table 29 on page 92

C

```
int SD_Module_getProductNameByIndex(int index, char *productName);
int SD_Module_getProductNameBySlot(int chassis, int slot, char* productName);
```

C++

```
int SD_Module::getProductName(int index, char *productName);
int SD_Module::getProductName(int chassis, int slot, char* productName);
```

LabVIEW

getProductName.vi

ProcessFlow

Available: No

3.2.4.5 getSerialNumber

This function returns the serial number of the specified device.

NOTE
Static Function: (Object-oriented languages only) getSerialNumber is a static function

Parameters

Name	Description
Inputs	
index	Module index. It must be in the range 0..(nModules-1), where nModules is returned by function *moduleCount*, Section 3.2.4.3 on page 83
chassis	Chassis number where the device is located. The chassis number can be found in Signadyne Manager (SDM) , or in nearly any Signadyne software. It can also be retrieved with the function *getChassis*, Section 3.2.4.6 on the following page
slot	Slot number where the device is plugged in. This number can be found on the chassis, in Signadyne Manager (SDM), or in nearly any Signadyne software. It can also be retrieved with the function *getSlot*, Section 3.2.4.7 on page 87
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
serialNumber	Serial number of the product. It can be used in function *open*, Section 3.2.4.1 on page 81
errorOut	See error codes in Table 29 on page 92

C

```
int SD_Module_getSerialNumberByIndex(int index, char *serialNumber);
int SD_Module_getSerialNumberBySlot(int chassis, int slot, char* serialNumber);
```

C++

```
int SD_Module::getSerialNumber(int index, char *serialNumber);
int SD_Module::getSerialNumber(int chassis, int slot, char* serialNumber);
```

LabVIEW

getSerialNumber.vi

ProcessFlow

Available: No

3.2.4.6 getChassis

This function returns the chassis number where the device is located.

NOTE
Static Function: (Object-oriented languages only) getChassis is a static function

Parameters

Name	Description
Inputs	
index	Module index. It must be in the range 0..(nModules-1), where nModules is returned by function *moduleCount*, Section 3.2.4.3 on page 83
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
chassis	Chassis number where the device is located. Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

```
int SD_Module_getChassis(int index);
```

C++

```
int SD_Module::getChassis(int index);
```

LabVIEW

getChassis.vi

ProcessFlow

Available: No

3.2.4.7 getSlot

This function returns the slot number where the device is located.

NOTE
Static Function: (Object-oriented languages only) getSlot is a static function

Parameters

Name	Description
Inputs	
index	Module index. It must be in the range 0..(nModules-1), where nModules is returned by function *moduleCount*, Section 3.2.4.3 on page 83
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
slot	Slot number where the device is plugged in. Negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	(LabVIEW only) See error codes in Table 29 on page 92

C

```
int SD_Module_getSlot(int index);
```

C++

```
int SD_Module::getSlot(int index);
```

LabVIEW

getSlot.vi

ProcessFlow

Available: No

3.2.4.8 PXItriggerWrite

This function sets the digital value of a PXI trigger in the PXI backplane. This function is only available in PXI / PXI Express form factors.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nPXItrigger	PXI trigger number
value	Digital value with negated logic, 0 (ON) or 1 (OFF)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
errorOut	See error codes in Table 29 on page 92

C

```
int SD_Module_PXItriggerWrite(int moduleID, int nPXItrigger, int value);
```

C++

```
int SD_Module::PXItriggerWrite(int nPXItrigger, int value);
```

LabVIEW

PXItriggerWrite.vi

3.2.4.9 PXItriggerRead

This function reads the digital value of a PXI trigger in the PXI backplane. This function is only available in PXI / PXI Express form factors.

Parameters

Name	Description
Inputs	
moduleID	(Non-object-oriented languages only) Module identifier, returned by function *open*, Section 3.2.4.1 on page 81
nPXItrigger	PXI trigger number
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
moduleIDOut	(LabVIEW only) A copy of moduleID
value	Digital value with negated logic, 0 (ON) or 1 (OFF), or negative numbers for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C

```
int SD_Module_PXItriggerRead(int moduleID, int nPXItrigger);
```

C++

```
int SD_Module::PXItriggerRead(int nPXItrigger);
```

LabVIEW

PXItriggerRead.vi

3.2.5 SD-Wave Class Functions

3.2.5.1 new

This function creates a waveform object from data points contained in an array in memory or in a file.

ADVANCED
Memory usage: Waveforms created with *New* are stored in the PC RAM, not in the module onboard RAM. Therefore, the limitation in the number of waveforms and their sizes is given by the amount of PC RAM.

Parameters

Name	Description
Inputs	
waveformType	Waveform Type (Table 18 on page 21)
waveformPoints	Number of points of the waveform
waveformDataA	Array with waveform points. Analog waveforms are defined with floating point numbers, which correspond to a normalized amplitude (-1 to 1)
waveformDataB	Array with waveform points, only for dual / IQ waveforms
waveformFile	File containing the waveform points
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
waveformID	(Non-object-oriented languages only) Waveform identifier, or a negative number for errors (see error codes in Table 29 on page 92)
errorOut	See error codes in Table 29 on page 92

C Function

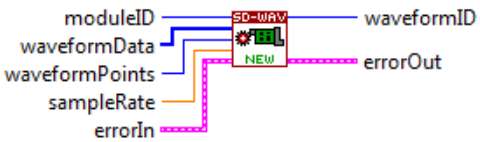
```
int SD_Wave_newFromArrayDouble(int waveformType, int waveformPoints, double* waveformDataA,
                               double* waveformDataB=0);
int SD_Wave_newFromFile(char* waveformFile);
```

C++ Function

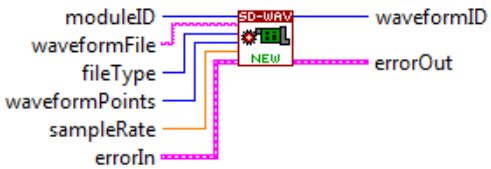
```
int SD_Wave::SD_Wave(int waveformType, int waveformPoints, double* waveformDataA, double* waveformDataB=0));
int SD_Wave::SD_Wave(char* waveformFile);
```

LabVIEW

SD_Wave_newFromArray.vi



SD_Wave_newFromFile.vi



3.2.5.2 delete

This function removes a waveform created with the *new* function.

ADVANCED
Onboard waveforms: Waveforms are removed from the PC RAM only, not from the module onboard RAM.

Parameters

Name	Description
Inputs	
waveformID	Waveform identifier (returned by *new*)
errorIn	(LabVIEW only) If it contains an error, the function will not be executed and errorIn will be passed to errorOut
Outputs	
errorOut	See error codes in Table 29 on the following page

C

```
int SD_Wave_delete(int waveformID);
```

C++

```
int SD_Wave::delete();
```

LabVIEW

delete.vi

3.2.6 Error Codes

Error Define	Error No	Error Description
SD_ERROR_OPENING_MODULE	-8000	Signadyne Error: Opening module
SD_ERROR_CLOSING_MODULE	-8001	Signadyne Error: Closing module
SD_ERROR_OPENING_HVI	-8002	Signadyne Error: Opening HVI
SD_ERROR_CLOSING_HVI	-8003	Signadyne Error: Closing HVI
SD_ERROR_MODULE_NOT_OPENED	-8004	Signadyne Error: Module not opened
SD_ERROR_MODULE_NOT_OPENED_BY_USER	-8005	Signadyne Error: Module not opened by user
SD_ERROR_MODULE_ALREADY_OPENED	-8006	Signadyne Error: Module already opened
SD_ERROR_HVI_NOT_OPENED	-8007	Signadyne Error: HVI not opened
SD_ERROR_INVALID_OBJECTID	-8008	Signadyne Error: Invalid ObjectID
SD_ERROR_INVALID_MODULEID	-8009	Signadyne Error: Invalid ModuleID
SD_ERROR_INVALID_MODULEUSERNAME	-8010	Signadyne Error: Invalid Module User Name
SD_ERROR_INVALID_HVIID	-8011	Signadyne Error: Invalid HVI
SD_ERROR_INVALID_OBJECT	-8012	Signadyne Error: Invalid Object
SD_ERROR_INVALID_NCHANNEL	-8013	Signadyne Error: Invalid channel number
SD_ERROR_BUS_DOES_NOT_EXIST	-8014	Signadyne Error: Bus doesn't exist
SD_ERROR_BITMAP_ASSIGNED_DOES_NOT_EXIST	-8015	Signadyne Error: Any input assigned to the bitMap does not exist
SD_ERROR_BUS_INVALID_SIZE	-8016	Signadyne Error: Input size does not fit on this bus
SD_ERROR_BUS_INVALID_DATA	-8017	Signadyne Error: Input data does not fit on this bus
SD_ERROR_INVALID_VALUE	-8018	Signadyne Error: Invalid value
SD_ERROR_CREATING_WAVE	-8019	Signadyne Error: Creating Waveform
SD_ERROR_NOT_VALID_PARAMETERS	-8020	Signadyne Error: Invalid Parameters
SD_ERROR_AWG	-8021	Signadyne Error: AWG function failed
SD_ERROR_DAQ_INVALID_FUNCTIONALITY	-8022	Signadyne Error: Invalid DAQ functionality
SD_ERROR_DAQ_POOL_ALREADY_RUNNING	-8023	Signadyne Error: DAQ buffer pool is already running

Table 29: Software error codes

References

[1] *Signadyne ProcessFlow, Hard Virtual Instrument (HVI) Programming Environment*. Product Website.

[2] *Signadyne VirtualKnob, Software Front Panels for Live Operation*. Product Website.

[3] *Signadyne Programming Libraries for Virtual Instrumentation*. Product Website.

SD AWG-H3300/H3300F Series
Arbitrary Waveform Generator with FPGA Option

Rev. 15.0424 (April 24, 2015)

Signadyne and its subsidiaries reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time.

Information delivered by Signadyne is believed to be accurate and reliable. However, no responsibility is assumed by Signadyne for its use, nor for any infringements of patents or other rights of third parties that may result from its use.

No license is granted by implication or otherwise under any patent or patent rights of Signadyne.

Trademarks and registered trademarks are the property of their respective owners.

www.signadyne.com