BIT-3000
Dynamic Sequencing Generator and Analyzer

User Manual 1.20

Notices

© BitifEye Digital Test Solutions GmbH 2018.

No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from BitifEye.

Edition

Nov 14, 2023

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, BitifEye disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. BitifEye shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should BitifEye and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to BitifEye's standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Product Labels

This electronic product is in compliance with the EMC and Safety regulations of the European Community.

Read the instructions in the BIT-3000 datasheet, as well as this manual, before using this electronic product.

This electronic product must be operated with AC power.

Technical Assistance

If you need product assistance or if you have suggestions, contact BitifEye. You will find the contact information on the BitifEye homepage at:

http://www.bitifeye.com

Representatives of BitifEye are available during standard German business hours.

Before you contact BitifEye, please note the actions you took before you experienced the problem. Then describe those actions and the problem to the technical support engineer.

Find a Mistake?

We encourage comments about this publication. Please report any mistakes to BitifEye (support@bitifeye.com)

Contents

List of Figures

Safety Precautions

WARNING	This manual describes the specifications and the proper usage of the BIT-3000 Dynamic Sequencing Generator and Analyzer (DSGA). All usage beyond the specifications or the intended use is not recommended and might cause impaired safety for the user.

Datasheet

This user manual does not cover specifications. Please also refer to the BIT-3000 Datasheet (available from the BitifEye website).

Intended Use

The BIT-3000 DSGA is intended to be used as test and measurement equipment.

This instrument is designed as laboratory equipment, and not intended for home use. The SMA and BNC connectors allow connection to appropriate electronic circuits or other laboratory equipment, as long as the specifications (see datasheet) are not violated.

The Ethernet and USB connectors allow to connect the BIT-3000 to standard computer equipment for the purpose of remote control, as well as for software updates.

Overview

The BIT-3000 DSGA consists of a frame, a clock module, and up to seven front-end modules. It is intended as a versatile protocol signal generator and detector.

1. Specifications

For specifications of the BIT-3000 DSGA, please refer to the datasheet.

2. Options

A BIT-3000 DSGA system consists of the following components:

- a mainframe
  - BIT-3000A or BIT-3000B Mainframe with FPGA, CPU, power supply, control unit
    - required
- a clock module
  - BIT-3001A or BIT-3001B Clock module
    - required
- up to seven front-end modules
  - BIT-3002A or BIT-3002B Dual differential generator module
    - optional (up to five per frame supported)
  - BIT-3003A or BIT-3003B Dual differential analyzer module
    - optional (up to two per frame supported)
  - BIT-3004A or BIT-3004B Dual single trigger in + dual single trigger out module
    - optional (up to two per frame supported)
  - BIT-30R22B Dual relay switch module
    - optional (up to seven per frame supported)
  - BIT-30S22B Differential solid-state switch module
    - optional (up to five per frame supported)

3. Usage

Initial Inspection

Before using the switch system, make sure that the following parts are included in the package:

- BIT-3000A or BIT-3000B Mainframe

- One BIT-3001A or BIT-3001B Clock Module, mounted in the Mainframe

- Up to seven front-end modules, mounted in the Mainframe (according to order)

Setup

The BIT-3000 DSGA can be operated as a desk-top instrument or can be mounted in a standard 19" rack. Make sure the specified environmental conditions are met.

Power

WARNING Only connect the AC connector at the rear side of the instrument to a power outlet using an appropriate 3-wire cable! Make sure you use a power outlet with a grounding terminal!

This device is sensitive to high voltage peaks. If there is a risk of power instability, it is recommended to add a UPS device to avoid any damage.

To turn the device on, flip the switch on the front side of the instrument to the position labeled "I". The blue LED next to the switch lights up to indicate the power state.

To turn the device off, flip the switch on the front side of the instrument to the position labeled "O".

Display

The display will light up as soon as the instrument is turned on, and it will display the FPGA firmware version after a few moments.

As soon as the firmware is ready, the display will change to "BIT-3000". If a network connection is available, the IP address will be displayed in the second line. If multiple IP addresses are available, the display will cycle through the available IPs.

During and after a system upgrade, the display will indicate the progress (see also page 28).

Maintenance cover

CAUTION This cover may only be removed on the instruction of BitifEye.

If the cover is removed without BitifEye's instruction or the seal is broken, the warranty for this device will be voided.

Connectors and Controls

Figure 1 shows the connectors and controls at the rear side of the instrument.
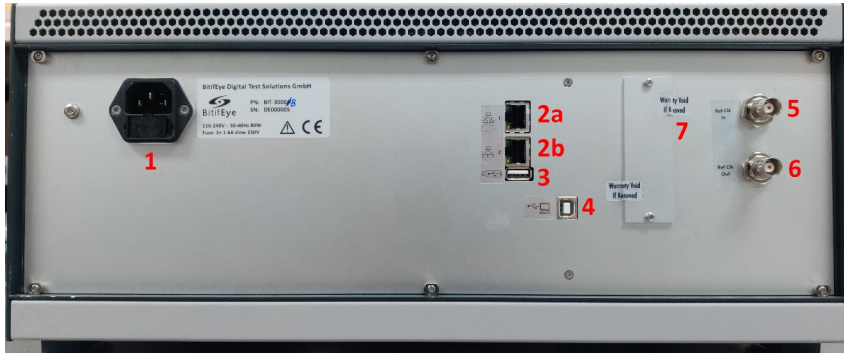
Figure 1: Connectors and Controls on Rear Side

1. Power inlet with fuse box (page 13)

2. Ethernet connectors 1 and 2 for remote control (page 23) and firmware updates (page 27)

3. USB connector for system upgrades (page 28)

4. USB connector for remote control (page 26)

5. Reference clock input (page 17), BNC

6. Reference clock output (page 17), BNC

7. Maintenance cover, sealed (page 13)

Figure 2 shows the connectors and controls at the front side of the instrument.

Figure 2: Connectors and Controls on Front Side

8. Generator module normal and complement outputs (page 18), SMA

9. Analyzer module normal and complement inputs (page 19), SMA

10. Trigger module output (page 19), SMA

11. Trigger module input (page 19), SMA

12. Relay module switch terminal (page 21), SMA

13. Solid-state switch module switch terminal (page 21), SMA

14. Solid-state switch trigger input (page 21), SMA

15. Display (page 13)

16. LAN reset (page 23)

17. Main power switch and indicator LED (page 13)

Cleaning

To clean any part of the BIT-3000 DSGA or its components, only use a dry cloth.

4. Architecture

This section explains the architecture of certain aspects of the BIT-3000 DSGA. It aims to help the user to understand the parameters that can be programmed.

System Overview
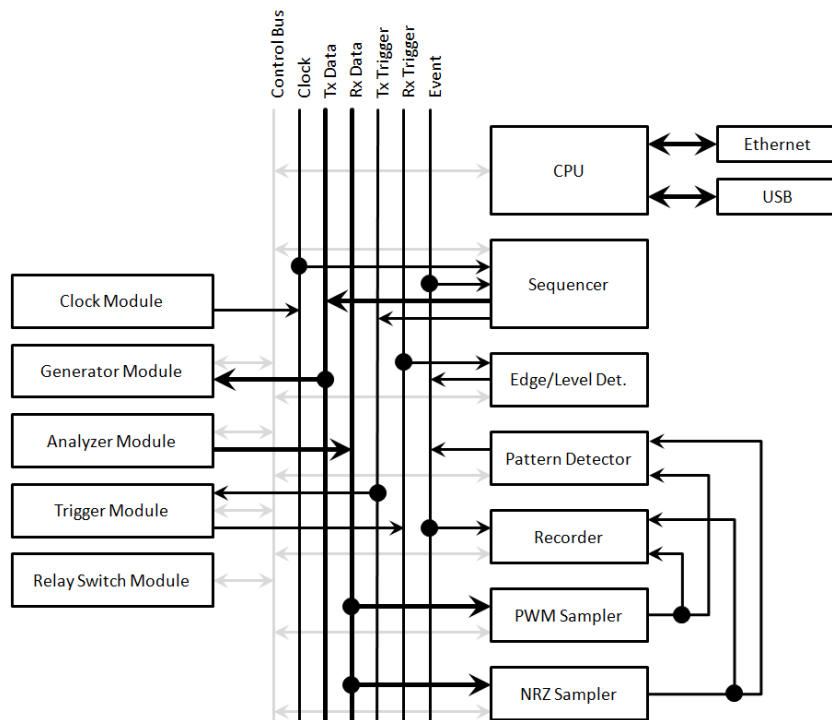
Figure 3: System Architecture

Accessible to the user are the clock module and any mounted generator, analyzer or trigger modules (Figure 3).

The clock is required for the sequencer, which can generate complex pattern streams, synchronized with trigger pulses. The patterns and trigger pulses can be used with the generators or trigger outputs.

Analyzers receive data streams that can be decoded by samplers (PWM or NRZ). The decoded pattern can be recorded, or they can be compared in order to fire events.

Events can also be fired by edges or levels detected by trigger inputs, or manually by the user. Those events can then be used for branch control in the sequencer, or for recording control in the pattern recorders.

The relay switches are controlled by the CPU directly.

Clock Module
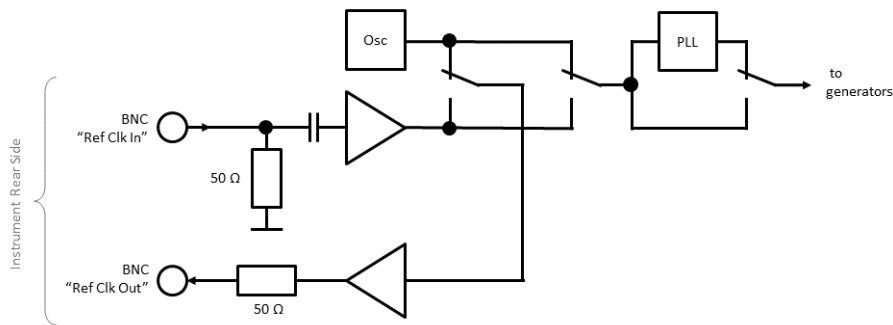
Figure 4: Clock Architecture

The DSGA has one clock module, which generates the reference clock for all generators and trigger outputs (Figure 4).

There is an internal reference clock, but the user can select to use an external clock instead. A PLL generates the requested system frequency.

The internal reference clock is output at another connector, which can also be switched to output the external reference clock (for daisy chaining).

Note that all switches shown in the figure can be operated independently.

The front-panel of the BIT-3001B clock module has a color LED. The LED is orange when the clock configuration is stopped due to an invalid configuration. The LED is red when the clock signal is unstable (e.g. when an external reference clock is missing or unstable).

Generator Module
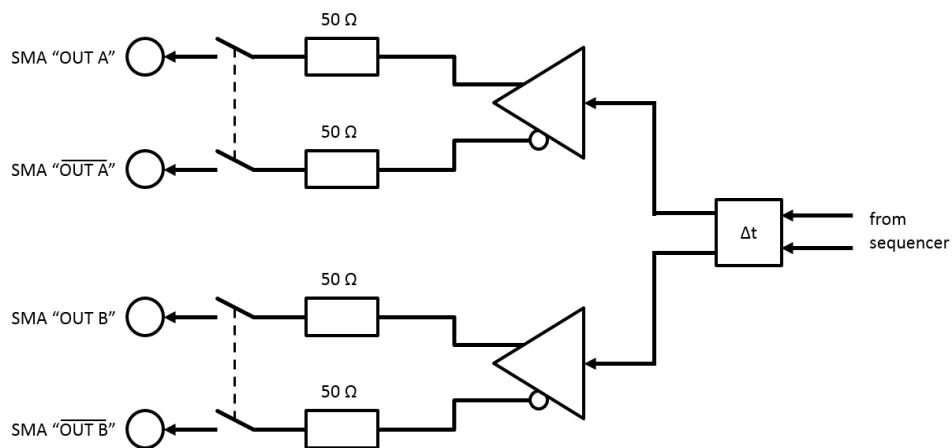
Figure 5: Generator Output Architecture

All generator outputs are differential (Figure 5). The swing voltage and the offset voltage can be programmed. The outputs have a 50 Ω impedance.

Note that the 50 Ω impedance is designed for the specified data rates of the BIT-3000; at higher frequencies, impedance mismatch is still possible.

A programmable delay can be applied to the signal. Note that both generators on a module share the same delay line; the data can be independent, but the delay is common.

Each output pair can be disconnected via a relay.

There is also an output protection circuit. If a high external DC voltage is applied, the output relays are turned off. It is not recommended that you rely on this mechanism in daily use; we advise the user to take care not to apply too high or too low DC voltages.

The front-panel of the generator module has two color LEDs, one for each differential generator output. When the LED is green, the output relay is enabled. When the LED is red, the output protection circuit disabled the relay due to a fault condition.

Analyzer Module
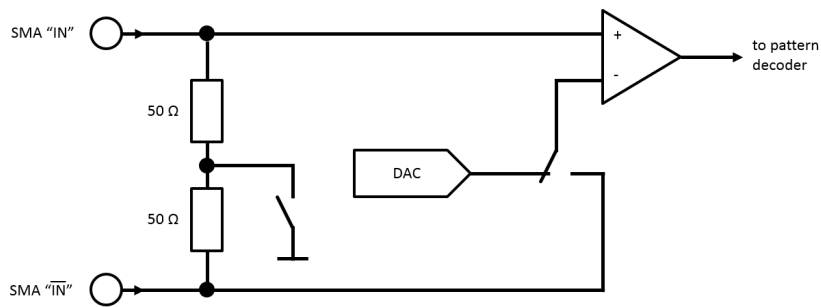
Figure 6: Analyzer Input Architecture

Each analyzer input has a comparator with a programmable threshold level. Instead of the programmable threshold, a second input signal can be connected.

Both inputs are connected together via two 50 Ω resistors. The middle net can be connected to ground via a programmable relay (Figure 6).

With this mechanism, four operation modes are possible:
- a single-ended signal is compared to a programmable threshold voltage; the input has high impedance; note that the complement input must not be left open, otherwise the connected source will see some termination resistance
- a single-ended signal is compared to a programmable threshold voltage; the input is terminated to ground
- a differential signal is connected directly to a comparator; the differential signal is terminated differentially with 100 Ω
- a differential signal is connected directly to a comparator; each signal is terminated with 50 Ω

Trigger Module
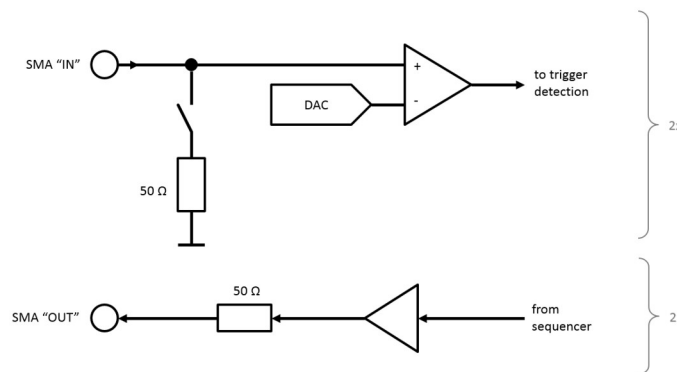
Figure 7: Trigger Input and Output Architecture

Each trigger input has a comparator with a programmable threshold level. The input has high impedance, but can be terminated via a 50 Ω resistor to ground using a programmable relay (Figure 7).

Note that the functionality of the trigger input on the trigger module is fundamentally different to the trigger inputs on the solid-state switch module. While the trigger input on the solid-state switch module controls only that solid-state switch, the trigger inputs on the trigger module allow to fire events, which in turn allow to control the sequencer and pattern recorders.

Each trigger output is a simple amplifier with a 50 Ω impedance.

The front-panel of the trigger module has four white LEDs (or blue for the A revision), one next to each terminal. Each LED individually flashes when the corresponding output or input signal changes. The timing is electronically adjusted to make very short pulses visible.

Relay Switch Module
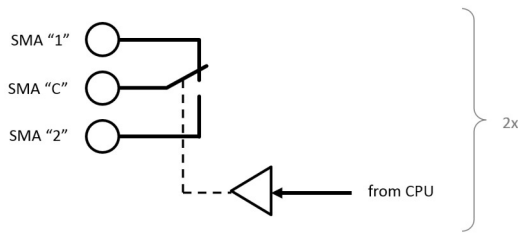
Figure 8: Relay Switch Architecture

Each relay switch has a center connector (labeled "C"), and two switchable connectors (labeled "1" and "2", respectively). The center connector can either be routed to "1" or to "2" (see figure 8); there's no all-open position.

The connector which is not routed to the center connector is open (i.e. reflective).

Compared to the solid-state switches, the relay switches provide superior RF performance, and can be operated with a wide DC range. However, switching from one signal to another is slower (in the millisecond range), and switch bouncing is a typical artifact.

The front-panel of the relay module has four green LEDs, one next to the "1" and "2" terminals of each switch. A LED is lit when the corresponding terminal is connected to the corresponding "C" terminal of the switch.

Solid-State Switch Module
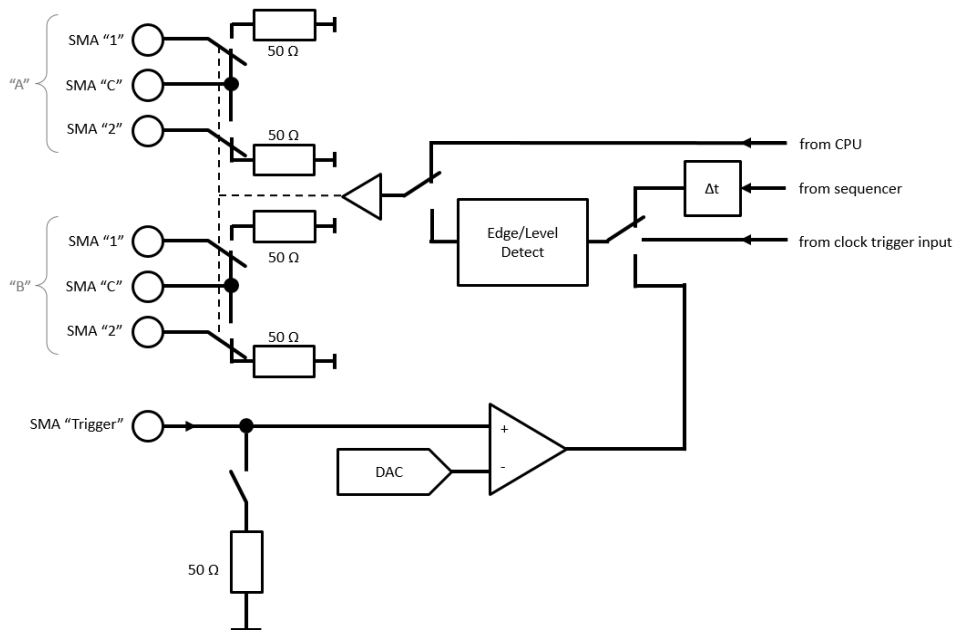
Figure 9: Solid-State Switch Architecture

Each module contains two solid-state switches, which are controlled by a common source.

Each solid-state switch has a center connector (labeled "C"), and two switchable connectors (labeled "1" and "2", respectively). The center connector can either be routed to "1" or to "2"; there's no all-open position.

The connector which is not routed to the center connector is terminated with 50 Ω into GND (i.e. absorptive); see figure 9.

The switches can be controlled by one of the following sources:

- Manually: the switch can be programmed into position "1" or "2" via the remote interface

- Trigger input: the trigger input on the module allows external instrument to control the switch position. The trigger works asynchronously to allow deterministic timing. The trigger can be programmed to control the switches directly, or through an edge-detector

- Clock module trigger: the trigger input on the clock module is routed to each solid-state switch module via a low-skew distribution network. This allows external instruments to control multiple solid-state switch modules in the same mainframe synchronously. The behavior is the same as for the local trigger input

- Sequencer: the solid-state switch module can be programmed to behave like a generator. In this mode, the data stream from a sequencer channel toggles the switch (bit "0" → path "1", bit "1" → path "2"). The timing can be adjusted with a programmable delay line.

The trigger input has a comparator with a programmable threshold level. The input has high impedance, but can be terminated via a 50 Ω resistor to ground using a programmable relay.

Note that the functionality of the trigger input on the solid-state switch module is fundamentally different to the trigger inputs on the trigger module. While the trigger inputs on the trigger module allow to fire events, which in turn allow to control the sequencer and pattern recorders, the trigger on the solid-state switch module controls only that solid-state switch.

Compared to the relay switches, the solid-state switches provide very short switch times (in the nanosecond range), they can be synchronized to the sequencer or external trigger sources, and provide very clean switching behavior. However, the bandwidth is lower compared to RF relays, and the switches are more sensitive to DC voltages and ESD.

The front-panel of the solid-state switch module has two push-buttons, which allow the user to temporarily override the configuration (which is normally programmed via the remote interface). Pushing the button next to the trigger temporarily forces the module to use the trigger input. Pushing the button again reverts to the previous configuration. Pushing the button between the switch terminals temporarily forces the switches to path "1". Pushing the button a 2nd time changes to path "2". Pushing the button a 3rd time reverts to the previous configuration.

The front-panel of the solid-state switch module has five LEDs. The white LED next to the trigger is lit or blinking when the trigger input is selected as the control source of the solid-state switches. A blinking LED indicates that this was enabled via the push-button. The four LEDs next to the "1" and "2" terminals of the solid-state switches are lit when the corresponding terminal is connected to the corresponding "C" terminal.

5. General Operation

Preface

The following instructions and feature descriptions apply to a BIT-3000 DSGA with firmware revision 1.12 or later.

VISA Connection

The BIT-3000 is intended to be programmed by sending SCPI command through a VISA connection.

To establish a VISA connection and start sending remote commands, you need to conduct the following steps:

- install appropriate software on your PC (see section "Keysight IO Libraries")

- connect the BIT-3000 DSGA to your PC, either through a network (see page 23) or via USB (see page 26)

- obtain the instrument's VISA address (also explained in the corresponding sections mentioned in pre previous point)

- connect to the instrument and send SCPI commands (see page 30)

Keysight IO Libraries

In order to have all necessary drivers and software, it is recommended to install the Keysight IO Libraries Suite. This manual uses the Keysight IO Libraries Suite 2018 as a reference.

Network Connection

You can establish a VISA socket or VXI-11 connection via the network.

Additionally, the instrument will create a locally accessible web page (see page 25), which provides information about the instrument, and allows to conduct firmware updates and system upgrades (see page 27).

Connecting to LAN

Connecting to a BIT-3000 DSGA can simply be done via Ethernet. You can connect over a company network, or directly to a PC. You can use cross-link cables as well as regular cables.

The instrument has two Ethernet connectors, labeled "1" and "2". The default configurations are:

- upper connector, "1": static IP 192.168.5.100/24

- lower connector, "2": DHCP and auto-IP

The current IP address will be shown on the instrument's display.

Most software applications will allow you to use a host name instead of an IP address, which might be more convenient for the user. Each BIT-3000 unit has an individual host name with the format "bit3000-*serial*.local", where "*serial*" is the serial number that you can find on the type plate of the instrument (see also figure 14). For example, if your instrument has serial number DE000042, you use the host name "bit3000-de000042.local" to access the instrument via a web browser or VISA connection.

The Ethernet configuration of each port can be changed, either using the web interface (see page 25), or remote commands (see page 73). You can bring the Ethernet connectors back into their default configuration by pressing and holding the "LAN Reset" push-button at the front side of the instrument for at least five seconds. The reset will be indicated on the display.

Keysight Connection Expert

The instrument should also be found by the Keysight Connection Expert (which is part of the Keysight IO Libraries Suite), as shown in Figure 10.
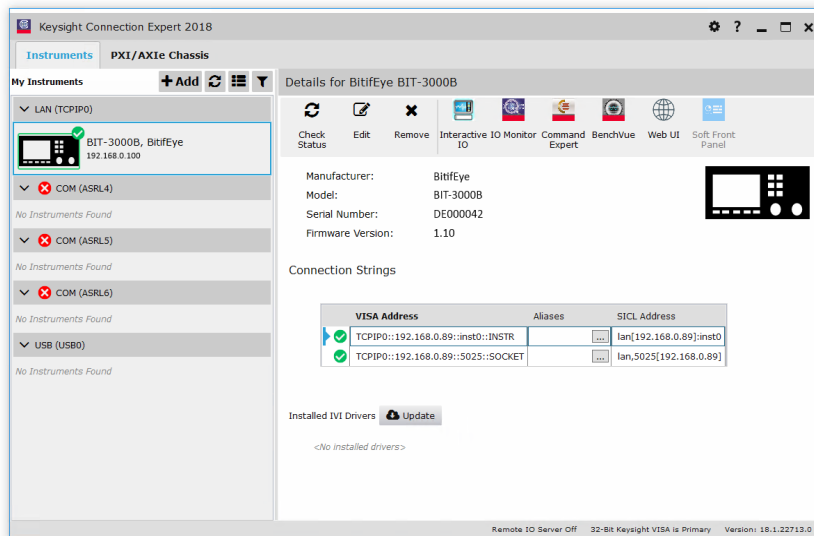
Figure 10: The BIT-3000 DSGA Shown in the Keysight Connection Expert

With Keysight Connection Expert 2018, the steps to add a BIT-3000 DSGA to the "My Instruments" list are:

- make sure the instrument is connected to the network properly

- in the Keysight Connection Expert, click "+ Add", then "LAN instrument"

- select your BIT-3000 DSGA unit from the list, click "OK"

Once it is in the "My Instruments" list, you can select it, and determine the VISA address, send SCPI commands, or access the instrument web page.

The VISA address can either be for the socket protocol, or for the VXI-11 protocol:

- Example for a socket address: "TCPIP::192.168.0.100::5025::SOCKET"

- Example for a VXI-11 address: "TCPIP::192.168.0.100::inst0::INSTR"

Most applications should work with either of the two protocols. You can also replace the IP that is encoded in the VISA address with the host name of the instrument.

Accessing the Web Interface

The web page of the BIT-3000 DSGA can be accessed from every web browser in the local network by entering the device's IP address or host name in the browser's address bar.
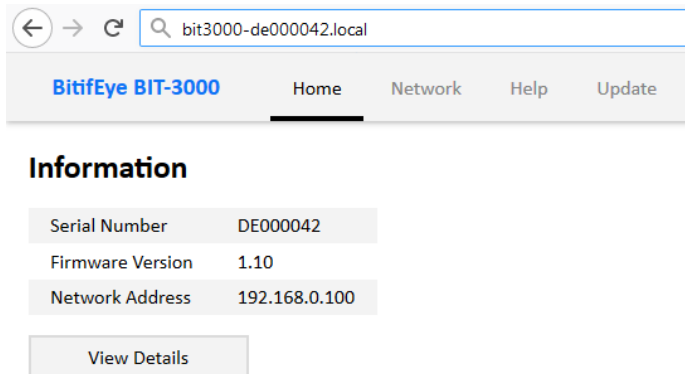
Figure 11: Instrument Web Page

Note that JavaScript is required for proper display of the web page.

Items you can access from the web page:

- Home: details about the instrument, e. g. serial numbers, firmware version, etc.
    - You can get more details about the modules (serial numbers, hardware revision, configware revision, etc.) by clicking "View Details"

- Network: change network settings, identify instrument (by making the display blink)

- Help: download user manual and data sheet, VISA addresses, software licenses

- Update: firmware update and system upgrade, see also page 27

USB Connection

The BIT-3000 DSGA can be controlled from any PC via a USB connection.

Note that there are two USB connectors on the rear side of the instrument. Only the square shaped (type B) connector can be used to establish a remote connection. The rectangular connector (type A) is only used for system upgrades (see page 28).

Connecting to USB

Connecting only requires a standard USB type A to USB type B connector cable.

When a BIT-3000 DSGA is connected to a PC for the first time, drivers are required. The BIT-3000 DSGA registers as a virtual serial port device. This driver is available from the BitifEye website.

Determining the VISA address

The VISA address is typically "ASRL#::INSTR", where "#" is the COM port address. The Keysight Connection Expert will automatically shown an item for this port under "My Instruments". Click "+ Add", "Serial Instrument on ASRL *portnumber*" to connect to the instrument.

If multiple ASRL ports (i.e. COM ports) are shown, you can determine the COM port address by opening the "device manager" in the Windows computer settings (see Figure 12). When the proper drivers are installed, one of the listed COM ports will be shown as "BitifEye BIT-3000".
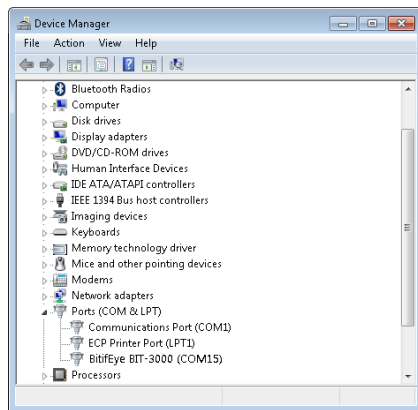
Figure 12: Virtual COM Port Address Shown in Device Manager

The Keysight Connection Expert will show a VISA address for the instrument, e.g. "ASRL10:INSTR". With this VISA address, a VISA connection can be established to send SCPI commands.

6. Software Updates

There are two kinds of software updates for the BIT-3000 DSGA:

- A "Firmware Update" provides the latest firmware to the instrument, providing latest bug-fixes and features

- A "System Upgrade" provides a new operating system for the instrument, which should rarely be necessary

To install a firmware update or a system upgrade, click the "Update" button on the menu panel of the web page.

CAUTION Only use appropriate update/upgrade files from BitifEye. Do not disconnect power during the update process.

Firmware Update

For a firmware update, an update file with the extension ".fwu", furnished by BitifEye, is required. Follow these steps:

- enter the update page on the instrument web page

- under "Firmware Update", select the .fwu-file, then click "Update"

- wait until the update is complete

Note that the instrument will be reset during this process, and all remote connections will be dropped.

During the firmware update, the instrument's display will show something like this (except during the first update after a system upgrade):

```
Updating...
192.168.0.100
```

Once the update is complete, you will see the regular message from the firmware:

```
BIT-3000
192.168.0.100
```

The web page will keep updating and shows the progress of the update. Once the web page reports that the update was successful, you can continue using your instrument as normal. No power-cycle is required.

System Upgrade

System upgrades allow new features to be provided that affect the underlying operating system that runs on the CPU of the BIT-3000 DSGA. A system upgrade file with the file name "image.stream", furnished by BitifEye, and a USB thumb drive, is required

Follow these steps:

- copy the file "image.stream" on a USB thumb drive

  - the USB thumb drive must contain exactly one partition which is formatted with FAT32 or ext4

  - the file must be in the top-level directory, with the filename unchanged

- plug the USB thumb drive into the USB type A connector at the rear side of the instrument

- if the instrument web page is accessible:

  - enter the update page on the instrument web page

  - under "System Upgrade", click Upgrade

  - follow the instructions on the screen

- otherwise:

  - turn the instrument off

  - press and hold the LAN reset button, and turn the instrument on

  - hold the LAN reset button until the display indicates that the system upgrade is in progress

- the instrument will reboot into a special upgrade system; you can follow the progress on the instrument's display, as the instrument web page won't be available during this process

- once the upgrade is done, the instrument will reboot into the freshly upgraded operating system

During the system upgrade, the instrument's display will show something like this:

```
Upgrading...
```

Once the update is complete, it should show something like this:

```
Upgrade Done
Rebooting...
```
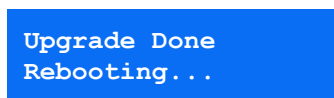
Note that as a next step, a firmware update is required; this will also be shown on the display:

```
FW-Upd. Required
192.168.0.100
```

Follow the instructions in the previous section (page 27) to conduct the firmware update.

If anything goes wrong

If a firmware update goes wrong, it is recommended that you read the displayed errors carefully. If possible, try to carry out the firmware update again. In the case of a failure, please send the reported errors to BitifEye support.

If a system upgrade fails, or if the system becomes unusable after a failed firmware update, you can repeat the system upgrade any time.

In the case of a failure, please send the errors reported on the instrument display to BitifEye support.

7. Remote Programming

This section describes the commands that are available for remote programming. Any language/framework that is capable of handling such instructions can be used for programming the BIT-3000 DSGA.

Alternatively, you can use the Keysight Interactive IO tool, which is part of the Keysight IO Libraries.

General Syntax

SCPI commands consist of the following parts:

- a header, which consists of mnemonics
  - each mnemonic is preceded by a colon, where the first colon is optional
  - a mnemonic may require an integer index, which is indicated by a `#` symbol in the documentation
- the header can be followed by a question mark, making it a query
- zero or more parameters
  - if any parameters are given, the first argument is preceded by a white-space
  - if multiple parameters are given, they are separated by commas

There are also a few IEEE-488 commands, which start with an asterisk instead of a semicolon.

Multiple commands can be concatenated into a transaction, by separating them with semicolons. However, note that the leading colon for subsequent commands is not optional anymore: if it is omitted, it is assumed that the preceding mnemonics are the same as for the first command.

Mnemonics are documented with an upper-case part and a lower-case part. You can either use only the letters in upper-case, or all letters together. The instrument handles the mnemonics case-insensitive.

Note that in this document, explicit line breaks are printed in C-syntax: line-feed characters (ASCII code 10) are printed as \n, carriage-return characters (ASCII code 13) are printed as \r.

Note that optional parts of the SCPI syntax are documented in square brackets `[ ]`, and parameters are documented in angle brackets `< >`. Parameter selections are documented with pipes `|` in between.

Data Formats

The following argument types are supported:

- strings
- block data
- integers
- floating-point numbers
- boolean flags
- keywords

Strings

Strings are always quoted with double-quotes (").

Example: `"some string"`.

Block Data

Block data (also known as "IEEE blocks") allows raw binary data to be transmitted. It begins with a hash symbol (#), followed by a digit indicating the number of decimal digits in the length of the payload, followed by the decimal digits of the length of the payload, followed by the payload in raw binary. Note that only data in multiples of eight bits can be represented.

Example: `#210UUUUUUUUU@`

The "2" in this example tells us that the length of the payload is a decimal number with two digits. Those two digits are "10", which tells us that the payload has exactly ten bytes (i.e. eighty bits). Those ten bytes are in the above example represented by their ASCII characters, which are "UUUUUUUUU@". Looking those up in an ASCII table yields the hexadecimal representation 0x5555555555555555540, which is the raw payload.

Integers

Plain integers.

Examples:

- `42`
- `-10`

Floating-Point Numbers

Floating point values are accepted as numbers with decimal places, optionally with an exponential suffix, and/or with an appended SI-prefix (kilo or mega; not case sensitive).

Examples:

- `3.141`
- `-5.2`
- `100e6`
- `10.2k`

Boolean Flags

Boolean flags can only be 0 or 1; alternatively, "on" or "off".

Examples:

- `0`
- `1`
- `off`
- `on`

Keywords

A keyword is similar to a string, except that it is not put into quotes, and only certain values are allowed. The accepted set of keywords is different for each command.

Examples:

- `SINGle`

- `DIFFerential`

When keywords are required as arguments, they can be given in either the short form or long form, just like mnemonics (e.g. the `SINGle` above could also be given as `SING`). When keywords are returned by a query, they are returned in the long form.

Module Numbering

Generator modules, analyzer modules and trigger modules can be mounted in any of the seven front-end module slots. These slots are numbered from one to seven.

However, in order to make remote programming easier, the modules themselves are not directly addressed; instead, the connectors are addressed. All connectors of a certain type are numbered starting from zero. The top connector of the left-most module has index zero. No numbers are skipped, even when a slot is empty of or a different type. Figure 13 illustrates the numbering scheme.
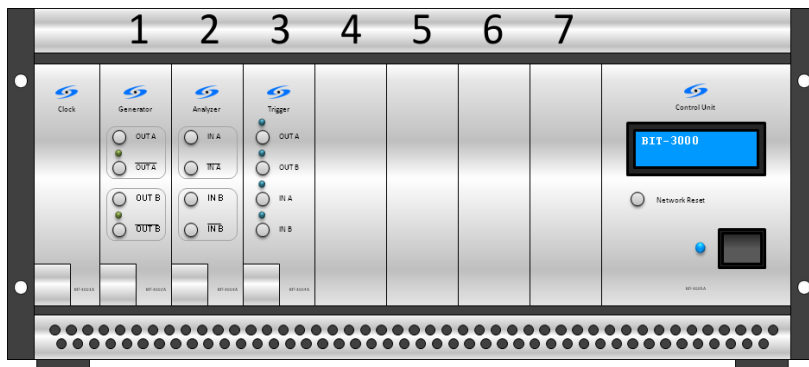
Figure 13: Module Numbering Scheme

IEEE-488 Commands

The following list explains all available IEEE-488 commands. Note that these commands must start with an asterisk (unlike regular SCPI commands), as shown in the syntax descriptions and in the examples.

*IDN?

Syntax: `*IDN?`

Returns the instrument's manufacturer, product number, serial number and firmware version. The values are separated by commas.

The manufacturer is "BitifEye", the product number is "BIT-3000A" or "BIT-3000B". The serial number consists of letters and digits. The firmware version consists of the major revision, a dot, and the minor revision.

Example query: `*IDN?`

Example response: `BitifEye,BIT-3000B,DE0000001,0.10`

*RST

Syntax: `*RST`

Resets the device into default state (the same state as after power-up).

Example command: `*RST`

*TST?

Syntax: `*TST?`

Conducts a self-test of all testable components of the instrument. The tests are the same tests that are conducted when the `:SYST:SELF?` Query (see page 34) is issued.

After the self-tests are complete, which might take a while, a numeric value is returned. If all self-tests pass, `0` is returned; otherwise, the number of errors is returned.

Note that the self-tests will take a while to complete, depending on how many and which modules are installed. You'll probably receive a time-out from your VISA client, unless the VISA timeout is programmed to 30 seconds or more.

If errors were detected, it is recommended to use the `:SYST:SELF?` query to get a human-readable report.

Example query: `*TST?`

Example response: `0`

Common Commands

:SYSTem:ERRor?

Syntax: `:SYSTem:ERRor?`

Returns the first error in the error queue. The error will be removed from the error queue.

The error message consists of an error code (integer) and an error message. When no error is in the queue, the error code is zero and the message is "No Error". The code and the message are separated by a comma.

Example query: `SYST:ERR?`

Example response: `0,"No Error"`

:SYSTem:ERRor:COUNt?

Syntax: `:SYSTem:ERRor:COUNt?`

Returns the number of errors in the error queue. The error queue will not be modified.

Example query: `SYST:ERR:COUN?`

Example response: `0`

:SYSTem:HELP:HEADers?

Syntax: `:SYSTem:HELP:HEADers?`

Returns a list of all supported SCPI headers, separated by carriage-return characters ("\r", see also page 30).

Example query: `SYST:HELP:HEAD?`

Example response: `"*IDN\r*RST\r:SYSTem:ERRor?\r:SYSTem:HELP:HEADers?"`
The actual response will contain many more headers. Depending on the software that you use, the "\r" character might be displayed as a new line or not.

:SYSTem:SELFtest?

Syntax: `:SYSTem:SELFtest?`

Conducts a self-test of all testable components of the instrument. The tests are the same tests that are conducted when the `*TST?` Query (see page 33) is issued.

After the self-tests are complete, which might take a while, a string is returned. If all self-tests pass, the string is `"pass"`; otherwise, the string starts with `"fail"`, followed by a human-readable description of the detected problems.

Note that the self-tests will take a while to complete, depending on how many and which modules are installed. You'll probably receive a time-out from your VISA client, unless the VISA timeout is programmed to 30 seconds or more.

Example query: `SYST:SELFtest?`

Example response: `"pass"`

:CONFiguration?

Syntax: `:CONFiguration?`

This command mainly exists for compatibility with the BIT-2100 Series Switch System. It enumerates all components of the instrument.

The returned string consists of a frame product number, followed by a colon and a space character, then the list of modules. The list of modules contains of product numbers, separated by a comma and a space character each, one item for each slot. The product number can also be "`empty`", in case the corresponding slot is empty.

Example query: `CONF?`

Example response: `"BIT-3000B: BIT-3001B, BIT-3002B, BIT-3003B, BIT-3004B, empty, empty, empty, empty"`

Clock Control

These commands control the central clock (see also page 17).

Note that the instrument will try to start the clock system immediately after a command was sent. However, this might fail under some circumstances. For example, if you switch to an external clock, you should set the clock frequency in the next step. However, this might fail, because the wrong frequency might be set between the two commands. In that case, the instrument postpones starting of the clock until it is necessary, e.g. when you try to start the sequencer, or when you try to configure generator parameters which require a stable clock to work. No error message is put into the error queue until starting the clock system fails when it is inevitable. If this behavior is undesired, you can force the clock system to start immediately with the `:CLOC:STAR` command.

:CLOCk:STARt

Syntax: `:CLOCk:STARt`

Forces an immediate start of the clock system. An error will be put into the error queue if starting the clock system is not possible (e.g. because an external reference clock is not stable).

Example command: `CLOC:STAR`

:CLOCk:FREQuency

Syntax: `:CLOCk:FREQuency <frequency>`

Sets the generated clock frequency. This clock frequency, in Hz, corresponds to the generator data bit range, in bits per second.

Example command: `CLOC:FREQ 100000000`

:CLOCk:FREQuency?

Syntax: `:CLOCk:FREQuency?`

Queries the currently programmed clock frequency.

Example query: `CLOC:FREQ?`

Example response: `100e6`

:CLOCk:SOURce

Syntax: `:CLOCk:SOURce <INTernal|EXTernal>`

Selects either the internal reference clock or the external reference clock (rear BNC clock input, see page 14). That clock is used as the reference clock for the PLL.

Example command: `CLOC:SOUR INT`

:CLOCk:SOURce?

Syntax: `:CLOCk:SOURce?`

Queries the currently selected clock source.

Example query: `CLOC:SOUR?`

Example response: `INTernal`

CLOCk:OUTPut:SOURce

Syntax: `:CLOCk:OUTPut:SOURce <INTernal|EXTernal>`

Selects the source for the external clock output connector at the rear of the instrument. Either internal (10 MHz reference oscillator; default) or external (the clock from the external clock input at the rear of the instrument, see page 14) can be selected.

Example command: `CLOC:OUTP:SOUR INT`

:CLOCk:OUTPut:SOURce?

Syntax: `:CLOCk:OUTPut:SOURce?`

Queries the currently selected source for the clock output at the rear of the instrument.

Example query: `CLOC:OUTP:SOUR?`

Example response: `INTernal`

:CLOCk:PLL:BYPass

Syntax: `:CLOCk:PLL:BYPass <bool>`

Enables or disables PLL bypassing. You can bypass the PLL when an external reference clock is used, so that this external reference clock serves are the system clock directly.

Example command: `CLOC:BYP 0`

:CLOCk:PLL:BYPass?

Syntax: `:CLOCk:PLL:BYPass?`

Queries the current state of the PLL bypassing feature.

Example query: `CLOC:PLL:BYP?`

Example response: `0`

:CLOCk:PLL:BANDwidth

Syntax: `:CLOCk:PLL:BANDwidth <LOW|HIGH>`

The PLL bandwidth can be set to either high or low. A high PLL bandwidth is recommended when low-frequency variations of the external reference clock must be tracked (e.g. SSC, which is typically modulated with 33 kHz). A low PLL bandwidth is recommended if minimum clock jitter is desired.

Example command: `CLOC:PLL:BAND LOW`

:CLOCk:PLL:BANDwidth?

Syntax: `:CLOCk:PLL:BANDwidth?`

Queries the current state of the PLL bandwidth.

Example query: `CLOC:PLL:BAND?`

Example response: `LOW`

:CLOCk:MULTiplier

Syntax: `:CLOCk:MULTiplier <multiplier>`

This parameter defines the factor the external reference clock is multiplied with.

Note that you must still provide the target frequency (using the `:CLOCk:FREQency` command); thus, the external reference clock frequency is implicitly defined by the system frequency, divided by the multiplier, multiplied by the divider (see `:CLOCk:DIVider`).

When the internal reference clock is selected, this parameter has no effect.

Example command: `CLOC:MULT 10`

:CLOCk:DIVider

Syntax: `:CLOCk:DIVider <multiplier>`

This parameter defines the divisor the external reference clock is divided by.

Note that you must still provide the target frequency (using the `:CLOCk:FREQency` command); thus, the external reference clock frequency is implicitly defined by the system frequency, multiplied by the divider, divided by the multiplier (see `:CLOCk:MULTiplier`).

When the internal reference clock is selected, this parameter has no effect.

Example command: `CLOC:DIV 10`

Sequencer Controls

The following SCPI commands control the sequencer, which is responsible for pattern generation.

:SEQuencer:PATTern:DOWNload

Syntax:
`:SEQuencer:PATTern:DOWNload <name>,<channel>,<data>`

This command allows you to download pattern data to the sequencer. The sequencer must be stopped before this command can be used. Each pattern must be given a name (quoted string format), a channel index (0 through 11), and the pattern data.

For a detailed description of the pattern data format, see section "Pattern Data Format" on page 80. The data can be either a string containing the characters `0` and `1` only, or block data.

Example command: `SEQ:PATT:DOWN "pattern1",0,#15abcde`

:SEQuencer:SEQuence:DOWNload

Syntax: `:SEQuencer:SEQuence:DOWNload <sequence>`

This command allows you to download a sequence to the instrument. The sequencer must be stopped before this command can be used.

For a detailed description of the sequence data format, see section "Sequence Data Format" on page 75.

Example command:
`SEQ:SEQ:DOWN "L1: PLAY a, 200, 0\nGOTO L1"`

:SEQuencer:CLEar

Syntax: `:SEQuencer:CLEar`

Deletes all patterns and the whole sequence. The sequencer must be stopped before this command can be used.

Example command: `SEQ:CLE`

:SEQuencer:RUN

Syntax: `:SEQuencer:RUN`

Starts the sequencer.

If the sequencer is configured for triggered start, the sequencer waits for the defined trigger.

If the sequencer is already running, this command has no effect.

Note that you must first download a sequence (see :SEQuencer:SEQuence:DOWNload), and all referenced patterns (see :SEQuencer:PATTern:DOWNload), before you can start the sequencer.

Example command: `SEQ:RUN`

:SEQuencer:STOP

Syntax: `:SEQuencer:STOP`

Stops the sequencer.

Example command: `SEQ:STOP`

:SEQuencer:CLOCkgenerator

Syntax: `:SEQuencer:CLOCkgenerator <divider>`

Configures the division factor for the divided clock generator feature. Only multiples of two are allowed.

Example command: `SEQ:CLOC 10`

:SEQuencer:STRobe

Syntax: `:SEQuencer:STRobe`

Strobes the "manual" event. This command can be used to make the sequencer branch if it uses that "manual" event.

Note that this command does the same as the command `:EVEN:STR "manual"` (see page 62). It will have no effect if the "manual" event is not used in the sequence. If any other component of the instrument is configured to use the "manual" event (e.g. a pattern recorder), it will also be affected by this command.

Example command: `SEQ:STR`

:SEQuencer:STRobe:BIT?

Syntax: `:SEQuencer:STRobe:BIT?`

Returns the bit index of the "manual" event (the same event that is affected by the `:SEQ:STR` command).

Note that this query does the same as the query `:EVEN:BIT? "manual"` (see page 60).

Example query: `SEQ:STR:BIT?`

Example response: `30`

:SEQuencer:STRobe:MASK?

Syntax: `:SEQuencer:STRobe:MASK?`

Returns the bit mask of the "manual" event (the same event that is affected by the `:SEQ:STR` command). If you use this bit mask in a BRAN instruction in the sequence (see page 76), the branch will be taken when the `:SEQ:STR` command is invoked.

Note that this query does the same as the query `:EVEN:MASK? "manual"` (see page 60). The query will fail if the "manual" event was deleted by a remote command.

Example query: `SEQ:STR:MASK?`

Example response: `1073741824`

:SEQuencer:STATe?

Syntax: `:SEQuencer:STATe?`

Returns the current sequencer state. Possible return values:

- `STOPped`: sequencer is stopped
- `WAITing`: sequencer was started, but is still waiting for the start trigger
- `RUNNing`: sequencer is running
- `ERRor`: an error occurred (most likely due to an unstable clock or a sequence with too short patterns)

Example query: `SEQ:STAT?`

Example response: `RUNNing`

:SEQuencer:STEP?

Syntax: `:SEQuencer:STEP?`

Returns the sequence step currently being played. The first instruction in the sequence corresponds to step 0.

If the sequencer is not running, –1 is returned.

Example query: `SEQ:STEP?`

Example response: `5`

:SEQuencer:CONDition

Syntax: `:SEQuencer:CONDition <IMMediate|TRIGgered>`

Selects between immediate sequence start or triggered start. Only edges or levels at trigger module inputs can trigger the sequence start.

If you need a sequence that triggers on a pattern, you can define a sequence that, for example, starts with all-zeros, and branches to the main sequence on a pattern event.

Example command: `SEQ:COND IMM`

:SEQuencer:CONDition?

Syntax: `:SEQuencer:CONDition?`

Returns the currently selected sequence start mode.

Example query: `SEQ:COND?`

Example response: `IMMediate`

:SEQuencer:CONDition:SOURce

Syntax: `:SEQuencer:CONDition:SOURce <source>`

Defines the source for the sequencer start trigger. Must be an identifier of a trigger input (from the `:TRIG:INP#:IDEN?` query).

Example command: `SEQ:COND:SOUR "TRIGGER0"`

:SEQuencer:CONDition:SOURce?

Syntax: `:SEQuencer:CONDition:SOURce?`

Queries the currently configured sequence start trigger source.

Example query: `SEQ:COND:SOUR?`

Example response: `"TRIGGER0"`

:SEQuencer:CONDition:LEVels:RISing

Syntax: `:SEQuencer:CONDition:LEVels:RISing <bool>`

Specifies whether the sequence start trigger is sensitive to the rising edge of the trigger input signal.

Example command: `SEQ:COND:LEV:RIS 1`

:SEQuencer:CONDition:LEVels:RISing?

Syntax: `:SEQuencer:CONDition:LEVels:RISing?`

Queries whether the sequence start trigger is sensitive to the rising edge of the trigger input signal.

Example query: `SEQ:COND:LEV:RIS?`

Example response: `1`

:SEQuencer:CONDition:LEVels:FALLing

Syntax: `:SEQuencer:CONDition:LEVels:FALLing <bool>`

Specifies whether the sequence start trigger is sensitive to the falling edge of the trigger input signal.

Example command: `SEQ:COND:LEV:FALL 1`

:SEQuencer:CONDition:LEVels:FALLing?

Syntax: `:SEQuencer:CONDition:LEVels:FALLing?`

Queries whether the sequence start trigger is sensitive to the falling edge of the trigger input signal.

Example query: `SEQ:COND:LEV:FALL?`

Example response: `1`

:SEQuencer:CONDition:LEVels:HIGH

Syntax: `:SEQuencer:CONDition:LEVels:HIGH <bool>`

Specifies whether the sequence start trigger is sensitive to a high level of the trigger input signal.

Example command: `SEQ:COND:LEV:HIGH 1`

:SEQuencer:CONDition:LEVels:HIGH?

Syntax: `:SEQuencer:CONDition:LEVels:HIGH?`

Queries whether the sequence start trigger is sensitive to a high level of the trigger input signal.

Example query: `SEQ:COND:LEV:HIGH?`

Example response: `1`

:SEQuencer:CONDition:LEVels:LOW

Syntax: `:SEQuencer:CONDition:LEVels:LOW <bool>`

Specifies whether the sequence start trigger is sensitive to a low level of the trigger input signal.

Example command: `SEQ:COND:LEV:LOW 1`

:SEQuencer:CONDition:LEVels:LOW?

Syntax: `:SEQuencer:CONDition:LEVels:LOW?`

Queries whether the sequence start trigger is sensitive to a low level of the trigger input signal.

Example query: `SEQ:COND:LEV:LOW?`

Example response: `1`

Generator Control

The following SCPI commands control the generator outputs (see also page 18).

:GENerator:COUNt?

Syntax: `:GENerator:COUNt?`

Returns the total number of available generator outputs.

Example query: `GEN:COUN?`

Example response: `2`

:GENerator#:OFFSet

Syntax: `:GENerator#:OFFSet <offset>`

Sets the DC offset voltage, in volts, of the generator output.

Example command: `GEN0:OFFS 0.2`

:GENerator#:OFFSet?

Syntax: `:GENerator#:OFFSet?`

Returns the currently programmed DC offset voltage, in volts, of the generator output.

Example query: `GEN0:OFFS?`

Example response: `200e-3`

:GENerator#:AMPLitude

Syntax: `:GENerator#:AMPLitude <amplitude>`

Sets the single-ended swing amplitude voltage, in volts, of the generator output.

If the generator uses PAM-4 encoding (see `:GEN#:ENC` on page 47), a 2nd amplitude can be defined by using the `:GEN#:AMPL:PAM` command instead (see page 44).

Example command: `GEN0:AMPL 0.5`

:GENerator#:AMPLitude?

Syntax: `:GENerator#:AMPLitude?`

Returns the currently programmed single-ended swing amplitude voltage, in volts, of the generator output.

If the generator uses PAM-4 encoding (see `:GEN#:ENC` on page 47), a 2nd amplitude can be queried by using the `:GEN#:AMPL:PAM?` query instead (see page 45).

Example query: `GEN0:AMPL?`

Example response: `500e-3`

:GENerator#:AMPLitude:PAMfour

Syntax: `:GENerator#:AMPLitude:PAMfour <amplitude>,<amplitude2>`

Sets the single-ended swing amplitude voltages, in volts, of the generator output. This command is intended for PAM-4 encoding, where two different amplitudes are in use. See `:GEN#:ENC` on page 47) for details.

Example command: `GEN0:AMPL:PAM 0.5,0.2`

:GENerator#:AMPLitude:PAMfour?

Syntax: `:GENerator#:AMPLitude:PAMfour?`

Returns the currently programmed single-ended swing amplitude voltages, in volts, of the generator output. This query is intended for PAM-4 encoding, where two different amplitudes are in use. See `:GEN#:ENC` on page 47) for details.

Example query: `GEN0:AMPL:PAM?`

Example response: `500e-3,200e-3`

:GENerator#:ENABle

Syntax: `:GENerator#:ENABle <bool>`

Enables or disables the generator output relay.

Example command: `GEN0:ENAB 1`

:GENerator#:ENABle?

Syntax: `:GENerator#:ENABle?`

Returns the current state of the output relay.

Example query: `GEN0:ENAB?`

Example response: `1`

:GENerator#:ERRor?

Syntax: `:GENerator#:ERRor?`

Queries the output protection state of the generator output. If an error was detected, this query returns `1`. To clear the error, the output relay must be turned on or off using the `:GEN#:ENAB` command.

Example query: `GEN0:ERR?`

Example response: `0`

:GENerator#:TERMination

Syntax: `:GENerator#:TERMination <OPEN|SINGle|DIFFerential>`

Specifies the termination scheme of the load that is connected to the generator output:

- `OPEN`: each generator output connects to a high-impedance load

- `SINGle`: each generator output connects to a load that is terminated with 50 Ω into ground, or into a termination voltage

- `DIFFerential`: the generator outputs are connected to a differential 100 Ω load, with no connection to ground

This setting is required to calculate the correct output levels.

Example command: `GEN0:TERM SING`

:GENerator#:TERMination?

Syntax: `:GENerator#:TERMination?`

Returns the currently configured terminations scheme.

Example query: `GEN0:TERM?`

Example response: `SINGle`

:GENerator#:VTERm

Syntax: `:GENerator#:VTERm <vterm>`

Defines the termination voltage of the connected load. This setting only has an effect if the termination scheme is set to `SINGle`.

This setting is required to calculate the correct output levels.

Example command: `GEN0:VTER 0.0`

:GENerator#:VTERm?

Syntax: `:GENerator#:VTERm?`

Returns the currently configured termination voltage.

Example query: `GEN0:VTER?`

Example response: `0`

:GENerator#:MODE

Syntax: `:GENerator#:MODE <DATapattern|DIVidedclock>`

Defines whether the generator output transmits data from the sequencer (`DATapattern`) or a divided clock signal (`DIVidedclock`).

Note that `DIVidedclock` implicitly sets the encoding to NRZ (see `:GEN#:ENC` on page 47).

Example command: `GEN0:MODE DAT`

:GENerator#:MODE?

Syntax: `:GENerator#:MODE?`

Returns the currently configured generator mode.

Example query: `GEN0:MODE?`

Example response: `DATapattern`

:GENerator#:ENCoding

Syntax: `:GENerator#:ENCoding <NRZ|PAM4>`

Defines whether the generator output uses NRZ encoding (`NRZ`) or PAM-4 encoding (`PAM4`).

The default is NRZ encoding: one sequencer channel sends a data stream to the generator output, offset and amplitude are adjustable. In PAM-4 mode, two sequencer channels send a data stream to the generator output, and there are two adjustable amplitudes.

When using PAM-4, the generator automatically uses the channel that was assigned by the `:GEN#:CHAN` command (see page 47), plus the next higher sequencer channel. The output levels are as follows:

Bit in assigned sequencer channel	Bit in next sequencer channel	Output level (normal)
0	0	Offset - Amplitude1
0	1	Offset + Amplitude1
1	0	Offset - Amplitude2
1	1	Offset + Amplitude2

The 2nd amplitude can be set with the `:GEN#:AMPL` command (see page 44).

Example command: `GEN0:ENC PAM4`

:GENerator#:ENCoding?

Syntax: `:GENerator#:ENCoding?`

Returns the currently configured generator encoding.

Example query: `GEN0:ENC?`

Example response: `NRZ`

:GENerator#:CHANnel

Syntax: `:GENerator#:CHANnel <channel>`

Defines the sequencer channel from which the generator gets its data pattern (only valid if the generator mode is set to `DATapattern`).

This channel number corresponds to the channel that was specified when downloading a pattern with the `:SEQ:PATT:DOWN` command.

By default, the 1st generator output is mapped to the 1st sequencer channel, etc. Note that you must re-assign the channels when using PAM-4 (see `:GEN0:ENC` on page 47), as you'll need two sequencer channels for each generator output.

Example command: `GEN0:CHAN 2`

:GENerator#:CHANnel?

Syntax: `:GENerator#:CHANnel?`

Returns the currently configured sequencer channel number.

Example query: `GEN0:CHAN?`

Example response: `2`

:GENerator#:SLOT?

Syntax: `:GENerator#:SLOT?`

Returns the slot number where the module is mounted which holds this generator output. The first front-end module slot has number 1.

Example query: `GEN0:SLOT?`

Example response: `3`

:GENerator#:CONNector?

Syntax: `:GENerator#:CONNector?`

Returns the output number of this generator output. The first output on a module has number 1.

Example query: `GEN0:CONN?`

Example response: `2`

:GENerator#:SERial?

Syntax: `:GENerator#:SERial?`

Returns the serial number of the module which holds this generator output.

Example query: `GEN0:SER?`

Example response: `"DE000001"`

:GENerator#:TYPE?

Syntax: `:GENerator#:TYPE?`

Returns the product type of the module which holds this generator output.

Example query: `GEN0:TYPE?`

Example response: `"BIT-3002B"`

Analyzer Control

The following SCPI commands control the generator outputs (see also page 19).

Note that even though the sampler configuration commands (be it PWM or NRZ) are located in a sub-tree of each independent analyzer input, there is only one global configuration for all PWM samplers and one global configuration for NRZ samplers. Therefore, changing the configuration (e.g. the PWM data rate) for one analyzer input will actually affect all analyzer inputs.

:ANAlyzer:COUNt?

Syntax: `:ANAlyzer:COUNt?`

Returns the total number of available analyzer inputs.

Example query: `ANA:COUN?`

Example response: `2`

:ANAlyzer#:IDENtifier?

Syntax: `:ANAlyzer#:IDENtifier?`

Returns the unique identifier for this trigger input. This identifier is required when defining events (see page 59).

Example query: `ANA0:IDEN?`

Example response: `"ANALYZER0"`

:ANAlyzer#:TERMinated

Syntax: `:ANAlyzer#:TERMinated <bool>`

Specifies whether the relay that connects the termination resistors to ground is enabled or not.

Note that in single-ended mode (see `#ANAlyzer#:MODE` command on page 51), the complement input must be left unconnected. See also section "Analyzer Module" on page 19.

Example command: `ANA0:TERM ON`

:ANAlyzer#:TERMinated?

Syntax: `:ANAlyzer#:TERMinated?`

Returns the current state of the termination relay.

Example query: `ANA0:TERM?`

Example response: `1`

:ANAlyzer#:THReshold

Syntax: `:ANAlyzer#:THReshold <threshold>`

Defines the threshold, in volts, of the input comparator.

This parameter has no effect if the input is configured to differential mode.

Example command: `ANA0:THR -0.2`

:ANAlyzer#:THReshold?

Syntax: `:ANAlyzer#:THReshold?`

Returns the currently programmed threshold level, in volts.

Example query: `ANA0:THR?`

`Example response:  -200e-3`

:ANAlyzer#:THReshold:AUTO

Syntax: `:ANAlyzer#:THReshold:AUTO <settlingTime>`

Sets its threshold, in volts, of the comparator automatically.

Note that the voltage is determined with an successive approximation method. The signal must have both high- and low-level states within the settling time. If the settling time argument is not provided, it is set to 1 ms. If your signal transitions slower than once per millisecond, you must provide an appropriate settling time.

The threshold has no effect if the input is configured to differential mode.

Note that this command requires an unused pattern trigger event (see also page 61). If all pattern trigger events are currently in use, this command will fail with an error.

Example command: `ANA0:THR:AUTO 10ms`

:ANAlyzer#:THReshold:AUTO?

Syntax: `:ANAlyzer#:THReshold:AUTO? <settlingTime>`

Measures the threshold with the same method as `ANA0:THR:AUTO`, but instead of programming the threshold to the comparator, it just returns the measured values.

Also check the notes of `ANA0:THR:AUTO`.

Two values are returned: the measured offset and the measured amplitude of the signal.

Example query: `ANA0:THR:AUTO? 10ms`

Example response: `0,500e-3`

:ANAlyzer#:MODE

Syntax: `:ANAlyzer#:MODE <SINGle|DIFFerential>`

Selects between single-ended input mode (`SINGle`) or differential input mode (`DIFFerential`).

In single-ended input mode, the programmed threshold voltage is used.

In differential input mode, the threshold voltage is not used; instead, the complement analyzer input is used as a reference.

Example command: `ANA0:MODE SING`

:ANAlyzer#:MODE?

Syntax: `:ANAlyzer#:MODE?`

Returns the currently configured input mode.

Example query: `ANA0:MODE?`

Example response: `SINGle`

:ANAlyzer#:SLOT?

Syntax: `:ANAlyzer#:SLOT?`

Returns the slot number where the module is mounted which holds this analyzer input. The first front-end module slot has number 1.

Example query: `ANA1:SLOT?`

Example response: `2`

:ANAlyzer#:CONNector?

Syntax: `:ANAlyzer#:CONNector?`

Returns the output number of this analyzer input. The first input connector on a module has number 1.

Example query: `ANA0:CONN?`

Example response: `1`

:ANAlyzer#:SERial?

Syntax: `:ANAlyzer#:SERial?`

Returns the serial number for the module which holds this analyzer input.

Example query: `ANA0:SER?`

Example response: `"DE000001"`

:ANAlyzer#:TYPE?

Syntax: `:ANAlyzer#:TYPE?`

Returns the product type of the module which holds this analyzer input.

Example query: `ANA0:TYPE?`

Example response: `"BIT-3003B"`

:ANAlyzer#:SAMPler:MODE

Syntax: `:ANAlyzer#:SAMPler:MODE <PWM|NRZ>`

Configures the mode of the pattern data sampler for this analyzer input to PWM or NRZ.

Example command: `ANA0:SAMP:MODE PWM`

:ANAlyzer#:SAMPler:MODE?

Syntax: `:ANAlyzer#:SAMPler:MODE?`

Returns the current pattern sampler mode (`PWM` or `NRZ`).

Example query: `ANA0:SAMP:MODE?`

Example response: `PWM`

:ANAlyzer#:SAMPler:PWM:RATE

Syntax: `:ANAlyzer#:SAMPler:PWM:RATE <rate>`

Defines the expected data bit rate, in bits per second, for the analyzer inputs that are configured to PWM mode.

Note that this is a global setting which affects all analyzer inputs that use PWM mode.

Example command: `ANA0:SAMP:PWM:RATE 1000000`

:ANAlyzer#:SAMPler:PWM:RATE?

Syntax: `:ANAlyzer#:SAMPler:PWM:RATE?`

Returns the data bit rate that was defined with the `:ANA#:SAMP:PWM:RATE` command.

Example query: `ANA0:SAMP:PWM:RATE?`

Example response: `1e6`

:ANAlyzer#:SAMPler:PWM:EDGE

Syntax: `:ANAlyzer#:SAMPler:PWM:EDGE <RISing|FALLing>`

Defines the edge with which the expected PWM data signal begins. This can be either `RISing` (the PWM data begins with a rising edge, and all rising edges are expected to be equidistant), or `FALLing` (the PWM data begins with a falling edge, and all falling edges are expected to be equidistant).

Note that this is a global setting which affects all analyzer inputs that use PWM mode.

Example command: `ANA0:SAMP:PWM:EDGE RIS`

:ANAlyzer#:SAMPler:PWM:EDGE?

Syntax: `:ANAlyzer#:SAMPler:PWM:EDGE?`

Returns the edge that was defined with the `:ANA#:SAMP:PWM:EDGE` command.

Example query: `ANA0:SAMP:PWM:EDGE?`

Example response: `RISing`

:ANAlyzer#:SAMPler:PWM:INVert

Syntax: `:ANAlyzer#:SAMPler:PWM:INVert <bool>`

Defines whether decoded PWM data has to be inverted bit-wise. The intended application is to adapt to different PWM coding schemes.

By default (inversion off), a long pulse encodes a logical one, and a short pulse encodes a logical zero. If inversion is enabled, a long pulse encodes a logical zero, and a short pulse encodes a logical one.

Note that this is a global setting which affects all analyzer inputs that use PWM mode.

Example command: `ANA0:SAMP:PWM:INV OFF`

:ANAlyzer#:SAMPler:PWM:INVert?

Syntax: `:ANAlyzer#:SAMPler:PWM:INVert?`

Queries the current PWM data inversion state.

Example query: `ANA0:SAMP:PWM:INV?`

Example response: `0`

:ANAlyzer#:SAMPler:NRZ:RATE

Syntax: `:ANAlyzer#:SAMPler:NRZ:RATE <rate>`

Specifies the expected data bit rate, in bits per second, of the input data when the analyzer input is configured to NRZ mode.

Note that this is a global setting which affects all analyzer inputs that use NRZ mode.

Example command: `ANA0:SAMP:NRZ:RATE 0.25e6`

:ANAlyzer#:SAMPler:NRZ:RATE?

Syntax: `:ANAlyzer#:SAMPler:NRZ:RATE?`

Returns the currently configured NRZ data bit rate, in bits per second.

Example query: `ANA0:SAMP:NRZ:RATE?`

Example response: `250e3`

:ANAlyzer#:SAMPler:NRZ:RUNLength:REQuire

Syntax: `:ANAlyzer#:SAMPler:NRZ:RUNLength:REQuire <runlength>`

Specifies the maximum run length of NRZ data that is required for the expected data.

Note that this command does not change anything on the decoder. All it does is throw an error (that can be queried with `:SYST:ERR?`) if the specified run length is longer than the decoder can handle.

Also note that even if this command succeeds without error, it cannot be guaranteed that data with the specified run length can be properly decoded. If jitter or other impairments are present in the signal, the decoding might also fail.

Example command: `ANA0:SAMP:NRZ:RUNL:REQ 4`

:ANAlyzer#:SAMPler:NRZ:RUNLength:MAXimum?

Syntax: `:ANAlyzer#:SAMPler:NRZ:RUNLength::MAXimum?`

Returns the maximum supported run length of NRZ data that the decoder can handle at the specified data rate.

Note that the return value is independent of the value that was written with the `:ANA#:SAMP:NRZ:RUNL:REQ` command.

Also note that even if the actual run length of the signal is within the range returned by this query, it cannot be guaranteed that the signal can be properly decoded. If jitter or other impairments are present in the signal, the decoding might also fail.

Example query: `ANA0:SAMP:NRZ:RUNL:MAX?`

Example response: `5`

Trigger Input Controls

These commands control the parameters of trigger inputs.

Trigger inputs are identified using a zero-based index. Input A of the left-most trigger output is indexed with zero.

:TRIGger:INPut:COUN?

Syntax: `:TRIGger:INPut:COUNt?`

Returns the number of available trigger inputs.

Example query: `TRIG:INP:COUN?`

Example response: `2`

:TRIGger:INPut#:IDENtifier?

Syntax: `:TRIGger:INPut#:IDENtifier?`

Returns the unique identifier for this trigger input. This identifier is required when defining events (see page 59), or when defining a triggered sequence start condition (see page 42).

Example query: `TRIG:INP0:IDEN?`

Example response: `"TRIGGER0"`

:TRIGger:INPut#:TERMinated

Syntax: `:TRIGger:INPut#:TERMinated <bool>`

Enables or disables the termination relay for this trigger input.

Example command: `TRIG:INP0:TERM ON`

:TRIGger:INPut#:TERMinated?

Syntax: `:TRIGger:INPut#:TERMinated?`

Returns the current state of the termination relay for this trigger input.

Example query: `TRIG:INP0:TERM?`

Example response: `1`

:TRIGger:INPut#:THReshold

Syntax: `:TRIGger:INPut#:THReshold <threshold>`

Sets the threshold level, in volts, for this trigger input.

Example command: `TRIG:INP0:THR -300e-3`

:TRIGger:INPut#:THReshold?

Syntax: `:TRIGger:INPut#:THReshold?`

Returns the currently programmed threshold level, in volts, for this input.

Example query: `TRIG:INP0:THR?`

Example response: `-300e-3:`

:TRIGger:INPut#:THReshold:AUTO

Syntax: `:TRIGger:INPut#:THReshold:AUTO <settlingTime>`

Sets its threshold, in volts, of the comparator automatically.

Note that the voltage is determined with an successive approximation method. The signal must have both high- and low-level states within the settling time. If the settling time argument is not provided, it is set to 1 ms. If your signal transitions slower than once per millisecond, you must provide an appropriate settling time.

The threshold has no effect if the input is configured to differential mode.

Note that this command requires an unused pattern trigger event (see also page 61). If all pattern trigger events are currently in use, this command will fail with an error.

Example command: `TRIG:INP0:THR:AUTO 10ms`

:TRIGger:INPut#:THReshold:AUTO?

Syntax: `:TRIGger:INPut#:THReshold:AUTO?`
Measures the threshold with the same method as `TRIG:INP0:THR:AUTO`, but instead of programming the threshold to the comparator, it just returns the measured values.

Also check the notes of `TRIG:INP0:THR:AUTO`.

Two values are returned: the measured offset and the measured amplitude of the signal.

Example query: `TRIG:INP0:THR:AUTO? 10ms`

Example response: `0,500e-3`

:TRIGger:INPut#:SLOT?

Syntax: `:TRIGger:INPut#:SLOT?`

Returns the slot number where the module is mounted which holds this trigger input. The first front-end module slot has number 1.

Example query: `TRIG:INP0:SLOT?`

Example response: `4`

:TRIGger:INPut#:CONNector?

Syntax: `:TRIGger:INPut#:CONNector?`

Returns the output number of this trigger input. The first input on a module has number 1.

Example query: `TRIG:INP0:CONN?`

Example response: `1`

:TRIGger:INPut#:SERial?

Syntax: `:TRIGger:INPut#:SERial?`

Returns the serial number of the module which holds this trigger input.

Example query: `TRIG:INP0:SER?`

Example response: `"DE000001"`

:TRIGger:INPut#:TYPE?

Syntax: `:TRIGger:INPut#:TYPE?`

Returns the product type of the module which holds this trigger input.

Example query: `TRIG:INP0:TYPE?`

Example response: `"BIT-3004B"`

Trigger Output Controls

These commands control the parameters of trigger outputs.

Trigger outputs are identified using a zero-based index. Output A of the left-most trigger output is indexed with zero.

Note that the trigger pulse length can only be programmed globally, not independently for each trigger output.

:TRIGger:OUTPut:COUNt?

Syntax: `:TRIGger:OUTPut:COUNt?`

Returns the number of available trigger outputs.

Example query: `TRIG:OUTP:COUN?`

Example response: `2`

:TRIGger:OUTPut:PULSe:LENGth

Syntax: `:TRIGger:OUTPut:PULSe:LENGth <seconds>`

Sets the desired pulse length for generated trigger pulses. The actual pulse length will be rounded to be an integer multiple of the system clock period.

Note that this parameter affects all trigger outputs. It cannot be programmed individually for each trigger output.

Example command: `TRIG:OUTP:PULS:LENG 1.0e-6`

:TRIGger:OUTPut:PULSe:LENGth?

Syntax: `:TRIGger:OUTPut:PULSe:LENGth?`

Returns the actual pulse length for generated trigger pulses. Note that this value might slightly differ from the value programmed with the `:TRIGger:OUTPut:PULSe:LENGth` command (see commend there).

Example query: `TRIG:OUTP:PULS:LENG?`

Example response: `1e-6`

:TRIGger:OUTPut#:POLartiy

Syntax: `:TRIGger:OUTPut#:POLarity <POSitive|NEGative>`

Sets the polarity of the trigger pulses that are generated by the specified trigger output. If positive polarity is set, the output idles low, otherwise it idles high.

Example command: `TRIG:OUTP0:POL NEG`

:TRIGger:OUTPut#:POLarity?

Syntax: `:TRIGger:OUTPut#:POLarity?`

Returns the polarity of the trigger pulses that are generated by the specified trigger output.

Example query: `TRIG:OUTP0:POL?`

Example response: `POSitive`

:TRIGger:OUTPut#:CHANnel

Syntax: `:TRIGger:OUTPut#:CHANnel <int>`

Assigns a trigger channel of the sequencer to this trigger output.

Example command: `TRIG:OUTP0:CHAN 1`

:TRIGger:OUTPut#:CHANnel?

Syntax: `:TRIGger:OUTPut#:CHANnel?`

Returns the currently assigned trigger channel of the sequencer for this trigger output.

Example query: `TRIG:OUTP0:CHAN?`

Example response: `1`

:TRIGger:OUTPut#:SLOT?

Syntax: `:TRIGger:OUTPut#:SLOT?`

Returns the slot number where the module is mounted which holds this trigger output. The first front-end module slot has number 1.

Example query: `TRIG:OUTP0:SLOT?`

Example response: `4`

:TRIGger:OUTPut#:CONNector?

Syntax: `:TRIGger:OUTPut#:CONNector?`

Returns the output number of this trigger output. The first output on a module has number 1.

Example query: `TRIG:OUTP0:CONN?`

Example response: `2`

:TRIGger:OUTPut#:SERial?

Syntax: `:TRIGger:OUTPut#:SERial?`

Returns the serial number of the module that holds this trigger output. Note that a trigger module has two trigger outputs, so there are always two trigger outputs that report the same serial number.

Example query: `TRIG:OUTP0:SER?`

Example response: `"DE000001"`

:TRIGger:OUTPut#:TYPE?

Syntax: `:TRIGger:OUTPut#:TYPE?`

Returns the product type of the module which holds this trigger output.

Example query: `TRIG:OUTP0:TYPE?`

Example response: `"BIT-3004B"`

Event Control

These commands define events that can be used to control sequencer branching or to control pattern recorders.

Each event can be identified by a user-provided string.

By default, the following two events are defined:

- "manual": can be manually strobed via commands (type: `MANual`)

- "immediate": is always applied (type: `IMMediate`)

These events can be modified just like any other event, but they cannot be deleted, and they will always be restored after reset.

:EVENts:COUNt?

Syntax: `:EVENts:COUNt?`

Returns the total number of defined events.

Example query: `EVEN:COUN?`

Example response: `2`

:EVENts:IDENtifier?

Syntax: `:EVENts:IDENtifier? <index>`

Returns the identifier string for a certain event. The event is in this case identified by a 0-based index.

Example query: `EVEN:IDEN? 0`

Example response: `"myevent"`

:EVENts:CLEar

Syntax: `:EVENts:CLEar [<id>]`

Deletes a specified event. The event to be deleted is specified by its identifier as a quoted string. If no event identifier is provided, all events are deleted.

The pre-defined events "manual" and "immediate" cannot be deleted.

When deleting an event, you are recommended to update all cases where the event is used (e.g. sequence, pattern recorders), because the behavior of a deleted event is undefined.

Example command: `EVEN:CLE "myevent"`

:EVENts:BIT?

Syntax: `:EVENts:BIT? <id>`

When using an event for branching in the sequence, a bit mask has to be used. This query returns the bit index (0-based) for a given event. The event is specified by a quoted string.

For example, if this query returns 5, you can use the mask 0x20 (= 32 decimal) in the sequence string to refer to this event. Note that it is not recommended that you continue to use this mask in the sequence after you have deleted the event.

Example query: `EVEN:BIT? "myevent"`

Example response: `0`

:EVENts:MASK?

Syntax: `:EVENts:MASK? <id>[,...]`

Similar to the `:EVEN:BIT?` query, this command returns a bit mask. One or more event identifiers, provided as quoted strings, can be given as an argument.

Note that the result value may be in the range from 0 to $2^{31}-1$, so an unsigned 32-bit integer is required.

Example query: `EVEN:MASK? "myevent","otherevent"`

Example response: `5`

:EVENts:TYPE

Syntax: `:EVENts:TYPE <id>,<type>`

Specifies the type of an event:

- `MANual`: the event can be manually strobed (using the `:EVEN:STR` command)
- `IMMediate`: the event fires continuously
- `LEVel`: the event is triggered on an edge or a level of a trigger input
- `PATTern`: the event is triggered on a pattern sampled at an analyzer input

In order to define a new event, just call this command. You can use any identifier (quoted string) that is not in use yet.

Example command: `EVEN:TYPE "myevent",PATT`

:EVENts:TYPE?

Syntax: `:EVENts:TYPE? <id>`

Returns the type of a certain event (identified by a quoted string).

Example query: `EVEN:TYPE? "myevent"`

Example response: `PATTern`

:EVENts:PATTern

Syntax: `:EVENts:PATTern <id>,<pattern>`

Specifies the pattern for a given event (identified by a quoted string). The pattern must be provided as a quoted string, in binary (i.e., only the characters `0` and `1` are allowed).

The pattern has no effect if the specified event is not configured to pattern mode.

Note that there is one pattern memory, which is segmented. When a pattern is specified which is longer than a single segment, subsequent segments are allocated to fulfill the memory requirement. This happens transparently to the user. However, when changing a pattern of an event to a pattern which requires more segments, an error is raised when the next segment is already in use by another event. One solution to this condition is to clear all pattern events, then defining all patterns new.

Example command: `EVEN:PATTern "myevent","011010"`

:EVENts:SOURce

Syntax: `:EVENts:SOURce <id>,<sourceid>`

Selects the source for a given event (identified as a quoted string). The source is provided as a quoted string; depending on the type of event, the source is a trigger input identifier (`:TRIG:INP#:IDEN?`) or an analyzer input identifier (`:ANA#:IDEN?`).

Example command: `EVEN:PATTern "myevent","ANALYZER0"`

:EVENts:SOURce?

Syntax: `:EVENts:SOURce? <id>`

Returns the currently selected source for an event.

Example query: `EVEN:SOUR? "myevent"`

Example response: `"ANALYZER0"`

:EVENts:LEVels:RISing

Syntax: `:EVENts:LEVels:RISing <id>,<bool>`

Specifies whether a pattern trigger event (identified by a quoted string) is sensitive to a rising signal edge.

Example command: `EVEN:LEV:RIS "myevent",ON`

:EVENts:LEVels:RISing?

Syntax: `:EVENts:LEVels:RISing? <id>`

Queries whether a pattern trigger event (identified by a quoted string) is sensitive to a rising signal edge.

Example query: `EVEN:LEV:RIS? "myevent"`

Example response: `1`

:EVENts:LEVels:FALLing

Syntax: `:EVENts:LEVels:FALLing <id>,<bool>`

Specifies whether a pattern trigger event (identified by a quoted string) is sensitive to a falling signal edge.

Example command: `EVEN:LEV:FALL "myevent",OFF`

:EVENts:LEVels:FALLing?

Syntax: `:EVENts:LEVels:FALLing? <id>`

Queries whether a pattern trigger event (identified by a quoted string) is sensitive to a falling signal edge.

Example command: `EVEN:LEV:FALL? "myevent"`

Example response: `0`

:EVENts:LEVels:HIGH

Syntax: `:EVENts:LEVels:HIGH <id>,<bool>`

Specifies whether a pattern trigger event (identified by a quoted string) is sensitive to a high signal level.

Example command: `EVEN:LEV:HIGH "myevent",ON`

:EVENts:LEVels:HIGH?

Syntax: `:EVENts:LEVels:HIGH? <id>`

Queries whether a pattern trigger event (identified by a quoted string) is sensitive to a high signal level.

Example query: `EVEN:LEV:HIGH? "myevent"`

Example response: `1`

:EVENts:LEVels:LOW

Syntax: `:EVENts:LEVels:LOW <id>,<bool>`

Specifies whether a pattern trigger event (identified by a quoted string) is sensitive to a low signal level.

Example command: `EVEN:LEV:LOW "myevent",OFF`

:EVENts:LEVels:LOW?

Syntax: `:EVENts:LEVels:LOW? <id>`

Queries whether a pattern trigger event (identified by a quoted string) is sensitive to a low signal level.

Example query: `EVEN:LEV:LOW? "myevent"`

Example response: `0`

:EVENts:STRobe

Syntax: `:EVENts:STRobe <id>`

Fires the event (identified by a quoted string) once.

Example command: `EVEN:STR "myevent"`

:EVENts:STATe:CURRent?

Syntax: `:EVENts:STATe:CURRent <id>`

Queries the current state of the event (identified by a quoted string). Returns `1` if the event is currently being fired.

For transient events, e.g. a rising-edge trigger, you are recommended to use the `:EVEN:STAT:LATC?` query instead.

Example query: `EVEN:STAT:CURR "myevent"`

Example response: `0`

:EVENts:STATe:LATChed?

Syntax: `:EVENts:STATe:LATChed? <id>`

Returns `1` if the event (identified by a quoted string) has been fired since the last query for this event.

Example query: `EVEN:STAT:LATC? "myevent"`

Example response: `1`

Pattern Recorder Control

These commands allow the pattern recorders to be configured and the recorded patterns to be retrieved.

:RECorder#:SOURce

Syntax: `:RECorder#:SOURce <sourceid>`

Specifies the analyzer from which this recorder gets its data. The analyzer is identified by a quoted string, which can be determined with the `:ANA#:IDEN?` query.

Example command: `REC0:SOUR "ANALYZER0"`

:RECorder#:SOURce?

Syntax: `:RECorder#:SOURce?`

Queries the currently assigned analyzer.

Example query: `REC0:SOUR?`

Example response: `"ANALYZER0"`

:RECorder#:EVENt

Syntax: `:RECorder#:EVENt <eventid>[,<eventid>[,...]]`

Defines which events trigger the pattern recorder to start recording. Events are identified via quoted strings.

Note that the events that you specify must be defined before this command is called (see e.g. page 83).

Example command: `REC0:EVEN "event1","event2"`

:RECorder#:EVENt:COUNt?

Syntax: `:RECorder#:EVENt:COUNt?`

Returns the number of events that are assigned to the pattern recorder.

Example query: `REC0:EVEN:COUN?`

Example response: `2`

:RECorder#:EVENt?

Syntax: `:RECorder#:EVENt? <index>`

Returns the identifier of an event that is assigned to the recorder. Since there can be multiple events assigned to a recorder, you can iterate over all assigned events using an index. To query the name of the first assigned event, use index 0. The index must be less than the number of events (see :RECorder#:EVENt:COUNt?).

Example query: `REC0:EVEN? 1`

Example response: `"event2"`

:RECorder#:RUN

Syntax: `:RECorder#:RUN <prebits>,<postbits>`

Starts the pattern recorder. Note that the recorder won't actually record anything unless an assigned event was fired.

If you want the recorder to start recording immediately after this command, you must assign it an immediate event (see page 60).

Example command: `REC0:RUN 100,20`

:RECorder#:STOP

Syntax: `:RECorder#:STOP`

Stops the pattern recorder.

Example command: `REC0:STOP`

:RECorder#:STATus?

Syntax: `:RECorder#:STATus?`

Returns the current state of the pattern recorder. Possible return values:

- `STOPped`: the recorder is currently not running
- `PREData`: the recorder has not triggered yet, i.e. it is still trying to acquire data before the trigger output
- `POSTdata`: the recorder has triggered, but hasn't recorded enough data yet, i.e. it is still trying to acquire data after the trigger point
- `DONE`: the recorder has finished recording, the pattern is ready to be downloaded

Example query: `REC0:STAT?`

Example response: `DONE`

:RECorder#:DOWNload?

Syntax: `:RECorder#:DOWN? [<BLOCkdata|BINarystring>]`

Downloads the recorded pattern. The pattern can be returned either as a quoted string in binary (only characters `0` and `1`) or as block data (see also section "Block Data" on page 31).

For binary data, the left-most bit in the returned string represents the bit that was first recorded. For block data, the MSB of the left-most character in the result represents the bit that was first recorded.

Example query: `REC0:DOWN? BIN`

Example response: `"010100001111001100001111"`

:RECorder#:DOWNload:BITS?

Syntax: `:RECorder#:DOWN:BITS?`

Returns the number of bits in the recorded pattern.

Example query: `REC0:DOWN:BITS?`

Example response: `1000`

Relay Switch Control

These commands control the relay switches.

Relay switches are identified using a zero-based index. Relay switch A of the left-most relay switch module is indexed with zero.

:RELay:COUNt?

Syntax: `:RELay:COUNt?`

Returns the number of available relay switches.

Example query: `REL:COUN?`

Example response: `2`

:RELay#:PATH

Syntax: `:REL#:PATH <int|bool>`

Sets the relay switch to the specified position. Either an integer or a boolean is accepted.

To make the connection "1" to "C", use the argument `0` or `off`.

To make the connection "2" to "C", use the argument `1` or `on`.

Example command: `REL0:PATH 1`

:RELay#:PATH:SYNChronous

Syntax: `:REL#:PATH:SYNC <int|bool>`

Normally you can only switch relays sequentially. This command allows you to switch both modules on a module synchronously. It sets the relay switch, as well as the other relay switch on the same module, to the specified position. Either an integer or a boolean is accepted.

Note that this command only ensures that the control signals for both relays on a module are asserted synchronously. Due to differences in the coils and the mechanics, the relays might still switch slightly out of sync.

Note that this feature requires a relay switch module with hardware revision 1.1 or newer, and a configware revision of 1.11 or newer. You can check the revisions via the web interface (see page 25).

To make the connections "1" to "C", use the argument `0` or `off`.

To make the connections "2" to "C", use the argument `1` or `on`.

Example command: `REL0:PATH:SYNC 1`
This example command will switch both relay 0 and relay 1 (as they are both in the same module).

:RELay#:PATH?

Syntax: `:REL#:PATH?`

Reports the position in which the relay switch currently is.

If the current relay switch connects "1" to "C", the return value is `0`.

If the current relay switch connects "2" to "C", the return value is `1`.

Example query: `REL0:PATH?`

Example response: `1`

:RELay#:SLOT?

Syntax: `:RELay#:SLOT?`

Returns the slot number where the module is mounted which holds this relay switch. The first front-end module slot has number 1.

Example query: `REL1:SLOT?`

Example response: `7`

:RELay#:CONNector?

Syntax: `:RELay#:CONNector?`

Returns the output number of this relay switch. The first output on a module has number 1.

Example query: `REL1:CONN?`

Example response: `2`

:RELay#:SERial?

Syntax: `:RELay#:SERial?`

Returns the serial number of the module that holds this relay switch. Note that a relay switch module has two relay switches, so there are always two relay switches that report the same serial number.

Example query: `REL0:SER?`

Example response: `"DE000001"`

:RELayt#:TYPE?

Syntax: `:RELay#:TYPE?`

Returns the product type of the module which holds this relay switch.

Example query: `REL0:TYPE?`

Example response: `"BIT-30R22B"`

Solid-State Switch Control

These commands control the solid-state switches.

Solid-state switches are identified using a zero-based index. The left-most solid-state switch module is indexed with zero.

Note that every solid-state switch module contains two solid-state switches which always switch synchronously. Therefore, each module only contains one logical switch.

:SOLidstateswitch:COUNt?

Syntax: `:SOLidstateswitch:COUNt?`

Returns the number of available solid-state switches.

Example query: `SOL:COUN?`

Example response: `2`

:SOLidstateswitch#:SLOT?

Syntax: `:SOLidstateswitch#:SLOT?`

Returns the slot number where the module is mounted which holds this solid-state switch. The first front-end module slot has number 1.

Example query: `SOL1:SLOT?`

Example response: `7`

:SOLidstateswitch#:CONNector?

Syntax: `:SOLidstateswitch#:CONNector?`

Returns the output number of this solid-state switch. The first output on a module has number 1.

Example query: `SOL1:CONN?`

Example response: `2`

:SOLidstatesiwtch#:SERial?

Syntax: `:SOLidstateswitch#:SERial?`

Returns the serial number of the module that holds this solid-state switch.

Example query: `SOL0:SER?`

Example response: `"DE000001"`

:SOLidstateswitch#:TYPE?

Syntax: `:SOLidstateswitch#:TYPE?`

Returns the product type of the module which holds this solid-state switch.

Example query: `SOL0:TYPE?`

Example response: `"BIT-30S22B"`

SOLidstateswitch#:SWITch:CONTrol

Syntax: `:SOLidstateswitch#:SWITch:CONTrol <source>`

Sets the control source for the solid-state switch. The following sources are available:

- `LOCaltrigger`: the switch is controlled by the trigger signal that is fed into the trigger input on the solid-state switch module itself

- `GLOBaltrigger`: the switch is controlled by the trigger signal that is fed into the trigger input on the clock module. This requires a BIT-3001B clock module and a BIT-3000B mainframe

- `MANual`: the switch is forced into a fixed position

- `SEQuencer`: the switch position is controlled by a data channel from the sequencer. The switch behaves like a generator channel, except that the data stream controls the switch position instead of a digital output

Note that when the user overrode the switch configuration via the push-buttons on the front panel, trying to use this command to change to another control source will fail. In that case, either the user must revert the override (using the push-buttons again), or by sending a reset command (`*RST`).

Example command: `SOL0:SWIT:CONT MAN`

SOLidstateswitch#:SWITch:CONTrol?

Syntax: `:SOLidstateswitch#:SWITch:CONTrol?`

Returns the currently configured control source for the solid-state switch. The returned value is one of the four keywords that are listed in the `:SOL:SWIT:CONT` command (`GLOBaltrigger`, `LOCaltrigger`, `MANual`, `SEQuencer`).

Note that the returned value might differ from the expected (programmed) value, which can happen if the user manually enables the local trigger input with the front panel buttons. See `:SOL#:SWIT:FRON:TRIG?` and `:SOL#:SWIT:FRON:PATH?` for more details.

Example query: `SOL0:SWIT:CONT?`

Example response: `MANual`

SOLidstateswitch#:SWITch:FRONtpaneloverride:TRIGger?

Syntax: `:SOLidstateswitch#:SWITch:FRONtpaneloverride:TRIGger?`

This query returns a boolean, which tells whether the user pushed the front-panel button to override the trigger into local trigger mode.

Example query: `SOL0:SWIT:FRON:TRIG?`

Example response: `0`

SOLidstateswitch#:SWITch:FRONtpaneloverride:PATH?

Syntax: `:SOLidstateswitch#:SWITch:FRONtpaneloverride:PATH?`

This query returns a boolean, which tells whether the user pushed the front-panel button to override the switch path.

Example query: `SOL0:SWIT:FRON:PATH?`

Example response: `0`

SOLidstateswitch#:SWITch:CONTrol:INVert

Syntax: `:SOLidstateswitch#:SWITch:CONTrol:INVert <bool>`

Enables or disabled inversion of the control signal.

Note that when the switch is programmed to be edge-sensitive, this means that the falling edge toggles.

Example command: `SOL0:SWIT:CONT:INV on`

SOLidstateswitch#:SWITch:CONTrol:INVert?

Syntax: `:SOLidstateswitch#:SWITch:CONTrol:INVert?`

Returns the current state of the control signal inversion.

Example query: `SOL0:SWIT:CONT:INV?`

Example response: `1`

SOLidstateswitch#:SWITch:CONTrol:SENSitivity

Syntax: `:SOLidstateswitch#:SWITch:CONTrol:SENSitivity <sensitivity>`

Sets the sensitivity mode of the switch with respect to the control signal:

- `EDGE`: switch is edge-sensitive; a rising edge of the control signal toggles the switch position
- `LEVel`: switch is level-sensitive; the switch position directly reflects the control signal

Example command: `SOL0:SWIT:CONT:SENS LEV`

SOLidstateswitch#:SWITch:CONTrol:SENSitivity?

Syntax: `:SOLidstateswitch#:SWITch:CONTrol:SENSitivity?`

Returns the current sensitivity mode of the switch with respect to the control signal.

Example query: `SOL0:SWIT:CONT:SENS?`

Example response: `LEVel`

SOLidstateswitch#:SWITch:SEQuencer:CHANnel

Syntax: `:SOLidstateswitch#:SWITch:SEQuencer:CHANnel <int>`

Select the sequencer channel that is used to control the solid-state switch. This setting is only used when the sequencer is selected as the control source of the switch.

Example command: `SOL0:SWIT:SEQ:CHAN 1`

SOLidstateswitch#:SWITch:SEQuencer:CHANnel?

Syntax: `:SOLidstateswitch#::SWITch:SEQuencer:CHANnel?`

Returns the currently selected sequencer channel which controls the solid-state switch (only when the sequencer is selected as control source of the switch).

Example query: `SOL0:SWIT:SEQ:CHAN?`

Example response: `1`

SOLidstateswitch#:SWITch:SEQuencer:DELay

Syntax: `:SOLidstateswitch#:SWITch:SEQuencer:DELay <seconds>`

Programs the delay (in seconds) of the control signal from the sequencer. This setting does not affect any other control source (global trigger input from clock module, local trigger input, or manual control).

Example command: `SOL0:SWIT:SEQ:DEL 5e-9`

SOLidstateswitch#:SWITch:SEQuencer:DELay?

Syntax: `:SOLidstateswitch#:SWITch:SEQuencer:DELay?`

Returns the currently programmed delay of the control signal from the sequencer.

Example query: `SOL0:SWIT:SEQ:DEL?`

Example response: `5e-9`

SOLidstateswitch#:SWITch:PATH

Syntax: `:SOLidstateswitch#:SWITch:PATH <int>`

Manually sets the position of the solid-state switch. The argument can be `0`, which means the connector labeled "C" is connected to the connector labeled "1", or `1`, which means the connector labeled "C" is connected to the connector labeled "2".

This command only works if the switch is configured for manual operation.

Note that when the user overrode the switch configuration via the push-buttons on the front panel, trying to use this command to change to another path will fail. In that case, either the user must revert the override (using the push-buttons again), or by sending a reset command (`*RST`).

Example command: `SOL0:SWIT:PATH 1`

SOLidstateswitch#:SWITch:PATH?

Syntax: `:SOLidstateswitch#:SWITch:PATH?`

Returns the current path of the solid-state switch (either `0` or `1`).

Note that the returned value might differ from the expected (programmed) value, which can happen if the user manually changes the switch path with the front panel buttons.

Example query: `SOL0:SWIT:PATH?`

Example response: `1`

SOLidstateswitch#:TRIGger:THReshold

Syntax: `:SOLidstateswitch#:TRIGger:THReshold <voltage>`

Sets the threshold voltage of the local trigger input.

Example command: `SOL0:TRIG:THR 100mV`

SOLidstateswitch#:TRIGger:THReshold?

Syntax: `:SOLidstateswitch#:TRIGger:THReshold?`

Returns the currently programmed threshold voltage of the local trigger input.

Example query: `SOL0:TRIG:THR?`

Example response: `0.1`

SOLidstateswitch#:TRIGger:TERMinated

Syntax: `:SOLidstateswitch#:TRIGger:TERMinated <bool>`

Enables or disables the 50 Ω termination for the local trigger input.

Example command: `SOL0:TRIG:TERM on`

SOLidstateswitch#:TRIGger:TERMinated?

Syntax: `:SOLidstateswitch#:TRIGger:TERMinated?`

Returns the state of the local trigger input termination.

Example query: `SOL0:TRIG:TERM?`

Example response: `1`

Network Configuration

The following SCPI commands control the two network interfaces of the instrument.

Note that there are two interfaces at the rear side of the instrument; the upper connector is addressed with index 1, the lower connector is addressed with index 2.

All addresses are IPv4.

:NETWork:RESet

Syntax: `:NETWork:RESet`

Resets both network interfaces into their default configurations (see also page 23).

Note that when you issue this command over a network connection, the connection will obviously drop.

Example command: `NETW:RES`

:NETWork:INTerface#:CONFigure

Syntax: `:NETWork:INTerface#:CONFigure <STATic|DHCP>[,<ip>, [<subnet>,]<gateway>]`

Configures a specified network interface to either DHCP or a static IP.

If using a static IP, the IP, a subnet mask and a gateway must be specified. Either the IP and a subnet mask are specified separately, or an IP with a subnet suffix must be specified.

Note that when you configure the network connection that you use to issue this command, the connection will obviously drop.

Example command:
`NETW:INT1:CONF STAT,"192.168.0.10/24","192.168.0.1"`

:NETWork:INTerface#:CONFigure:MODE?

Syntax: `:NETWork:INTerface#:CONFigure:MODE?`

Returns the configured mode (`STATic` or `DHCP`) of the network port.

Example query: `NETW:INT1:CONF:MODE?`

Example response: `STATic`

:NETWork:INTerface#:CONFigure:IP?

Syntax: `:NETWork:INTerface#:CONFigure:IP?`

Returns the configured IP of the network port.

Note that this IP is not necessarily the IP under which the instrument can be reached (e.g. when no DHCP address is assigned). To determine the actual IP, use the `:NETW:INT#:IP?` query.

Example query: `NETW:INT1:CONF:IP?`

Example response: `"192.168.0.10"`

:NETWork:INTerface#:CONFigure:SUBNetmask?

Syntax: `:NETWork:INTerface#:CONFigure:SUBNetmask?`

Returns the configured subnet mask of the network port.

Example query: `NETW:INT1:CONF:SUBN?`

Example response: `"255.255.255.0"`

:NETWork:INTerface#:CONFigure:GATeway?

Syntax: `:NETWork:INTerface#:CONFigure:GATeway?`

Returns the configured gateway IP of the network port.

Example query: `NETW:INT1:CONF:GAT?`

Example response: `"192.168.0.1"`

:NETWork:INTerface#:IP?

Syntax: `:NETWork:INTerface#:IP?`

Returns the actual IP of the network port. If the network interface is inactive, the IP 0.0.0.0 is returned.

You can use this query for instance when you connect via USB and want to determine the IP address of the instrument (if you have physical access to the instrument, you can also just read the IP from the LC display).

Example query: `NETW:INT1:IP?`

Example response: `"192.168.0.1"`

:NETWork:IP?

Syntax: `:NETWork:IP?`

Returns the actual IP of the network in general. If both network ports have a valid IP, the address of the first port is returned. If none of the network interfaces are active, the IP 0.0.0.0 is returned.

You can use this query for instance when you connect via USB and want to determine the IP address of the instrument (if you have physical access to the instrument, you can also just read the IP from the LC display).

Example query: `NETW:IP?`

Example response: `"192.168.0.1"`

Sequence Data Format

The sequence is defined using a simple, assembly-language-like language. Each command can be preceded by a label, which must be followed by a colon.

Note that both pattern names and label names must consist only of Latin letters, digits, and underscores. The first character must not be a digit.

When using the BitifEye Frame Generator software, a higher-level pattern description language can be used.

PLAY Instruction

Syntax: `PLAY <pattern>, <length>[, <triggerout>]`

Plays the specified number of bits of a pattern.

If optional trigger bits are provided (a bit-mask), the corresponding trigger outputs are strobed.

Example sequence excerpt:

```
PLAY pat1, 100
```

This example plays the first 100 bits of the pattern "pat1"

LOOP Instruction

Syntax: `LOOP <level>, <count>, <label>`

Branches back to a provided label until a certain loop count is reached. A loop level (0-based) must be provided. If you do not have nested loops, you may set the level to 0 for all loops. If you use nested loops, each loop must have a different level than any of its outer loops.

Note that a loop count of 1 means that the branch is never taken, a loop count of 2 means that the branch is taken once, etc.

Example sequence excerpt:

```
label1: PLAY pat1, 100
        LOOP 0, 10, label1
```

This example plays the pattern "pat1" (which has 100 bits) ten times.

Note that in this example, if the data bit rate is e.g. 1 Mbps, the total play time will be 1 millisecond (100 bits multiplied by 10, divided by 1,000,000 bits per second). This way you can generate delays.

BRAN Instruction

Syntax:
```
BRAN [!]<conditions>, <label>[, <loopclearbits>]
```

Branches to a certain label if any of the events encoded by conditions bit-mask occurred. If multiple bits are set to 1 in that mask, the branch is taken if either of the events occurred (logical OR). The whole condition can be inverted using an exclamation mark, which means the branch is taken when none of the events occurred (i.e. events are latched). Note that the branch is taken if the event has occurred at any time since that event was used the last time.

When the branch is taken, the corresponding event latches are cleared. This means that a later branch instruction that is sensitive to the same event mask will only be taken if the event occurred again between the two branches. You can manually clear the event latches using the `CLTR` instruction (see page 76).

You can determine the bit mask for one or multiple events using the `:EVENts:MASK?` query (see page 60). If you just want to use an event that you can trigger manually, use the mask returned by `:SEQ:STR:MASK?` (see page 40).

When branching out of a loop, the corresponding loop counters are left in an undefined state. To fix this, when branching out of a loop, the corresponding bits (e.g. the LSB for the first loop level) must be explicitly provided.

Example sequence excerpt:
```
label1: PLAY pat1, 100
        BRAN !0b1, label1
```

This example plays the pattern "pat1" (which has 100 bits) *until* the 1st (lowest) event bit is set.

Another example sequence excerpt:
```
label1: PLAY pat1, 100
        BRAN 0b10, label1
```

This example plays the pattern "pat1" (which has 100 bits) *while* the 2nd event bit is set.

GOTO Instruction

Syntax: `GOTO <label>[, <loopclearbits>]`

Unconditionally branches to a label. The loop-clear-bits have the same meaning as for the BRAN command.

Example sequence:

```
label1: PLAY pat1, 100
        GOTO label1
```

This example plays the pattern "pat1" (which has 100 bits) in an infinite loop.

CLTR Instruction

Syntax: `CLTR <triggerclearbits>`

Resets latched events. This means that a branch defined by a subsequent BRAN command is not taken if the corresponding event bit was reset using the CLTR command.

Example sequence excerpt:

```
CLTR 0b111
```

This example clears the three lowest event latch bits.

One example where there `CLTR` instruction is useful if you want to make sure that an event is only taken into account if it occurred *after* a certain point in the sequence.

For example, the following sequence excerpt will play pattern "pat1" first, and will then loop "pat2" until the first event (mask 0b1) occurred (regardless of *when* it happened), after which it plays "pat3":

```
        PLAY pat1, 100
label1: PLAY pat2, 100
        BRAN !0b1, label1
        PLAY pat3, 100
```

If you want to make sure that "pat3" is only played if the event happened *after* "pat1", you can use the CLTR instruction to clear the trigger:

```
        PLAY pat1, 100
        CLTR 0b1
label1: PLAY pat2, 100
        BRAN !0b1, label1
        PLAY pat3, 100
```

Limitations

The sequence can consist of up to 512 instructions.

The pattern memory can be split into as many independent patterns as needed. The individual patterns don't have any granularity restriction, but a minimum pattern length is required. If this minimum pattern length is violated, the sequencer might enter an error state (refer to the `ERRor` response of the `:SEQ:STAT?` query on page 41).

The minimum pattern length depends on the data bit rate, and the number of loops and branches that reach the corresponding PLAY command. The minimum pattern length for a data rate of 100 MBit/s is:

Location of PLAY Instruction	Minimum Pattern Length
isolated or reached by a single loop	20 bit
reached by up to 3 loops + branches	40 bit
reached by up to 4 loops + branches	50 bit
reached by up to 5 loops + branches	60 bit
reached by up to 6 loops + branches	70 bit
reached by up to 7 loops + branches	80 bit
reached by up to 8 loops + branches	90 bit

Sequence Data Examples

The following example just plays the pattern "pattern1", which is 100 bits in length, in a loop.

```
start: PLAY pattern1, 100
       GOTO start
```

Note that "pattern1" doesn't necessarily have to have exactly 100 bits; it is only required that the pattern is at least as long as the number of bits specified in the sequence.

The next sequence example plays pattern "pattern1" again, but this time a trigger pulse is generated when the pattern starts. After that pattern, "pattern2" is played ten times. Then the sequence starts over.

```
start: PLAY pattern1, 100, 1
loop:  PLAY pattern2, 100
       LOOP 1, 10, loop
       GOTO start
```

The next sequence example plays "pattern1" (100 bits) in a loop. When the condition mask is met (i.e. at least bit 0 is set), "pattern2" is played once (also 100 bits), then the sequence starts over again.

```
start: PLAY pattern1, 100
       BRAN !1, start
       PLAY pattern2, 100
       GOTO start
```

In the following example, "pattern1" is played in a loop until the condition mask "1" (i.e. at least bit 0 is set) is met. The "pattern2" is played in a loop until the condition mask "8" (i.e. at least bit 3 is set) is met. Thus, "pattern1" and "pattern2" are played in an alternating way, with the two conditions switching between the patterns.

```
pat1:  PLAY pattern1, 100
       BRAN 1, pat2
       PLAY pattern1, 100
       GOTO pat1
pat2:  PLAY pattern2, 100
       BRAN 8, pat1
       PLAY pattern2, 100
       GOTO pat2
```

If you wanted to trigger the branch manually (i.e. using a remote command), you could simply use the mask returned by the query `:SEQ:STR:MASK?`, and then trigger the event once using the command `:SEQ:STR` (this uses the "manual" event; see page 59).

In the final example, "pattern1" is played 1000 times, then "pattern2" is played. Unless the condition mask "1" is met (i.e. only bit 0 is set), the sequence loops back to the beginning; otherwise (i.e. bit 0 is not set), "pattern3" is played in between.

Note the "CLTR" instruction. With that instruction, the branch to "pattern3" is only taken if the condition bit was set while "pattern2" was playing, as the trigger conditions are reset before that pattern starts playing.

```
start: PLAY pattern1, 100
       LOOP 1, 1000, start
       CLTR 1
       PLAY pattern2, 100
       BRAN !1, start
loop:  PLAY pattern3, 100
       GOTO start
```

Pattern Data Format

Each pattern is defined via a name, a channel index, and the pattern data.

The name must consist only of Latin letters, digits, and underscores. The first character must not be a digit. The pattern can be referenced in the sequence using that name.

The channel index is a 0-based index. By default, index 0 maps to the first generator output.

Pattern data can be provided either as a string or as as block data.

If provided as a string, the string must contain only the characters `1` and `0`. The left-most character is transmitted first.

Example: `"00001111"`

If provided as block data, the most significant bit of the left-most symbol is transmitted first.

Example: `#15abcde`

For more details about block data, see section "Block Data" on page 31.

In this example, the raw binary payload is "abcde"; hexadecimal representation of that ASCII data stream is 0x6162636465, or in binary 0b 0110000101100010011000110110010001100101. Therefore, the first transmitted bit will be a 0, followed by a 1, another 1, a 0, etc.

Programming Examples

Best Practice

It is generally recommended that you query the error queue after every command or query. The advantages of this approach are:

- when an error occurs, it's easier to find out which command or query triggered the error

- it can be verified that all commands are actually executed (which is not the case with syntax or argument errors, for example)

- the remote control software is always in sync with the instrument's firmware, as the query delays the remote control software until the command is completely handled and checked

The following programming examples do not show any error queries, for the sake of brevity and readability. You are recommended to use the following pseudo-code to run the programming examples, or any remote programming:

- command:

  1. send command string plus termination character

  2. send error query string plus termination character

  3. read and handle response

  4. repeat steps 2–4 until no error is reported

- query:

  1. send query string plus termination character

  2. read and handle response

  3. send error query string plus termination character

  4. read and handle response

  5. repeat steps 3–5 until no error is reported

Generating Patterns

The following example sequence of SCPI commands generates a pattern of 80 Mbit/s on the first generator output, with a swing of 1 Vp-p around zero.

```
*RST
:GEN0:AMPL 1
:CLOC:FREQ 80e6
:SEQ:PATT:DOWN "pat1",0,#15PPPPP
:SEQ:SEQ:DOWN "start: PLAY pat1,40\nGOTO start"
:SEQ:RUN
:GEN0:ENAB 1
```

In ASCII, the character "P" encodes the value 0x50, so the pattern will be 0101000001010000...

The following example additionally programs the first trigger output to generate a pulse each time the pattern repeats. Also, the second generator output will generate a 10 MHz clock signal.

```
*RST
:GEN0:AMPL 1;:GEN1:AMPL 1
:CLOC:FREQ 80e6
:SEQ:CLOC 8
:SEQ:PATT:DOWN "pat1",0,#15PPPPP
:SEQ:SEQ:DOWN "start: PLAY pat1,40,1\nGOTO start"
:SEQ:RUN
:GEN1:MODE DIV
:GEN0:ENAB 1;:GEN1:ENAB 1
```

Recording Data

The following example programs the analyzers to detect a 10 Mbit/s PWM-encoded signal (threshold 150 mV) on the first analyzer input. 100 bits of the pattern are recorded and downloaded.

```
:ANA0:THR 0.0
:ANA0:MODE SING
:ANA0:SAMP:MODE NRZ
:ANA0:SAMP:NRZ:RATE 10e6
:ANA0:IDEN?
:REC0:SOUR "ANALYZER0"
:REC0:EVEN "immediate"
:REC0:RUN 50,50
:REC0:STAT?
:REC0:DOWN:BITS?
:REC0:DOWN? BIN
```

The event "immediate" doesn't need to be explicitly defined (see section "Event Control" on page 59). By tying the recorder to that event, it will immediately start recording after the run command.

Note that in a real application, the response from the `:ANA0:IDEN` query would be used as an argument for the `:REC0:SOUR` command. Also, the `:REC0:STAT?` query would be repeated until the result becomes `1`.

The response from the `:REC0:DOWN?` query is the recorded pattern. It might be longer than the specified 100 bits (i.e. 50 before the trigger, and 50 after the trigger). The pattern is in this case returned as a string of binary characters (`0` and `1`).

Defining Events

The previous example used the "immediate" event. The next example will define a custom event that fires when a specific pattern is observed.

```
*RST
:ANA0:THR 0.0
:ANA0:MODE SING
:ANA0:SAMP:MODE NRZ
:ANA0:SAMP:NRZ:RATE 10e6
:EVEN:TYPE "patterntrigger",PATT
:EVEN:SOUR "patterntrigger","ANALYZER0"
:EVEN:PATT "patterntrigger","1110111"
:ANA0:IDEN?
:REC0:SOUR "ANALYZER0"
:REC0:EVEN "patterntrigger"
:REC0:RUN 10,10
:REC0:STAT?
:REC0:DOWN:BITS?
:REC0:DOWN? BIN
```

The event "patterntrigger" uses the same data source (analyzer 0 sampler), and fires when the pattern "1110111" is observed. The pattern recorder will record at least ten bits before that pattern is observed, and at least ten bits after. Therefore, the pattern "1110111" will always appear in the recorded pattern.

Using Events in the Sequencer

This example shows how to define an event (a rising edge trigger) and how to use this event in the sequence.

First, the event is defined, and its mask is queried:

```
*RST
:TRIG:INP0:IDEN?
:EVEN:TYPE "risingedge",LEV
:EVEN:SOUR "risingedge","TRIGGER0"
:EVEN:LEV:RIS "risingedge",ON
:EVEN:LEV:FALL "risingedge",OFF
:EVEN:LEV:HIGH "risingedge",OFF
:EVEN:LEV:LOW "risingedge",OFF
:EVEN:MASK? "risingedge"
```

The query `:TRIG:INP0:IDEN?` is required to determine the identifier of the trigger module input (it is "TRIGGER0").

The query `:EVEN:MASK? "risingedge"` returns the value 1, i.e. only bit 0 is set. This mask can now be used in the sequencer, for example in this sequence:

```
start: PLAY pattern1, 100
       BRAN !1, start
       PLAY pattern2, 100
       GOTO start
```

In this case, the event mask ("1") is inverted, i.e. the branch is taken if the event did not fire. Therefore, the sequencer plays "pattern1" in a loop, until the event (rising edge on trigger input 0) is detected.

Generating Electrical Idle

Some standards require an "electrical idle" state, i.e. the differential output amplitude is zero. The PAM-4 capabilities of the generator can be exploited to achieve this.

The following example sequence of SCPI commands generates a pattern of 50 Mbit/s on the first generator output, with a 500 mVp-p signal, offset 250 mV, and electrical idle, on the 2nd generator output:

```
*RST
:GEN1:ENC PAM4
:GEN1:CHAN 1
:GEN1:OFFS 0.25
:GEN1:AMPL:PAM 0.5;0
:CLOC:FREQ 50e6
:SEQ:PATT:DOWN "testpat",1,"01001111010111100000"
:SEQ:PATT:DOWN "testpat",2,"00000000000000001111"
:SEQ:SEQ:DOWN "start: PLAY testpat,20\nGOTO start"
:SEQ:RUN
:GEN1:ENAB 1
```

In this example, the 2nd sequencer channel was arbitrarily chosen. Since PAM-4 is employed, the 3rd channel is implicitly also mapped to the generator. The two patterns (channel 1 and 2) look like this:

Channel 1	`01001111010111100000`
Channel 2	`00000000000000001111`
Generator Output	`LHLLHHHHLHLHHHHLIIII`

When the 3rd channel (i.e. channel 2) is 1, the 2nd output amplitude is used. Since the 2nd amplitude is programmed to zero, the generator will output electrical idle. In the above table, the three possible output states are indicated as "`L`" (low), "`H`" (high) and "`I`" (idle).

Theoretically a 4th state would be possible (with the pattern bits being "`1`" on channel 1, and "`1`" on channel 2, but this would result in an output of an inverted bit with amplitude zero, which is indistinguishable from the former electrical idle state.

8. User Serviceable Parts

Changing Fuses

WARNING   Before changing fuses, make sure the device is physically disconnected from AC power! Never operate the BIT-3000 DSGA with the fuses or the fuse carrier removed!

To change the fuses, follow these instructions:

1. Make sure the device is powered off and all cables are disconnected

2. Open the fuse carrier on the rear side of the instrument (see Figure 14)

    1. release the fuse carrier using a screwdriver

    2. pull the fuse carrier out

3. Replace both fuses with 1.6AT 250 V slow blow

4. Insert and close the fuse carrier properly (see Figure 15)

Figure 14: Removing Fuses

Figure 15: Inserting Fuses

Replacing Modules

WARNING Before replacing a module, make sure the device is physically disconnected from AC power! Never operate the BIT-3000 DSGA while a module slot is unequipped! Never remove the Control Unit! The lid of the Control Unit covers components carrying AC power.

CAUTION The clock module should not be removed, as it has RF cables attached which can easily tangle up or get bent.

To remove a front-end module, follow these instructions:

1. Make sure the device is powered off and all cables are disconnected

2. Open the four Phillips screws on the top and bottom of the module (see Figure 16)

3. Eject the module only by pulling the ejection lever as shown on the photo (see Figure 17)

4. Remove the module by sliding it outwards

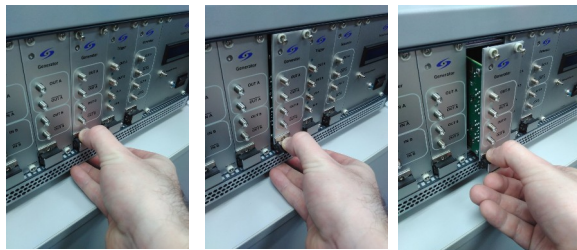**Figure 16: Module screws**

Figure 17: Removing a module

To insert a new front-end module, follow these steps:

1. Make sure the device is powered off and all cables are disconnected

2. Insert the module into the red guiding rails

3. Push it into the frame, only by pushing the ejection lever (see Figure 18)

   • Apply force only to the ejection lever! Do not push the module by pressing against parts of the front panel!

   • Make sure the module's front panel does not overlap with neighboring modules' front panels, and make sure the RF gasket (the soft metal fabric at the right side of the module) doesn't get crushed!

4. Tighten the four Phillips screws on the top and bottom of the module (see Figure 16)
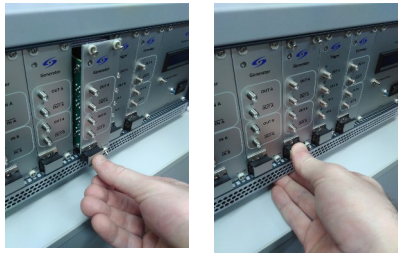

Figure 18: Inserting a module

Removing Modules

WARNING Before removing a module, make sure the device is physically disconnected from AC power! Never operate the BIT-3000 DSGA while a module slot is unequipped!

To remove a module, follow the module removal instructions in the section "Replacing Modules" on page 87. Then, close the slot using a filler panel.

Inserting Modules

WARNING Before inserting a module, make sure the device is physically disconnected from AC power! Never operate the BIT-3000 DSGA while a module slot is unequipped!

To insert a new module, remove the filler panel from the slot, then follow the module insertion instructions in the section "Replacing Modules" on page 87.

Note that since filler panels do not have handles, it may be easier to remove the panel by removing one of the adjacent modules first.

9. List of Acronyms

BNC   Bayonet Neill Concelman (connector)

CPU   Central Processing Unit

DAC   Digital-to-Analog Converter

DHCP   Dynamic Host Configuration Protocol

DSGA   Dynamic Sequencing Generator and Analyzer

FPGA   Field-Programmable Gate Array

IP   Internet Protocol

LAN   Local Area Network

LC(D)   Liquid Crystal (Display)

LED   Light-Emitting Diode

LSB   Least Significant Bit

LXI   LAN-based eXtensions for Instrumentation

MSB   Most-Significant Bit

RZ   Non-Return to Zero

PC   Personal Computer

PLL   Phase-Locked Loop

PWM   Pulse Width Modulation

RF   Radio Frequency

SCPI   Standard Commands for Programmable Instruments

SMA   Sub-Miniature A (Connector)

USB   Universal Serial Bus

VISA   Virtual Instrument Software Architecture