Runtime Programming of Speedster FPGAs (AN025)

Achronix
Data Acceleration

June 22, 2021 **Application Note**

Introduction

A Speedster®7t FPGA is configured at startup using a supplied bitstream containing the user design and the configuration of the interface subsystems in the I/O ring within the FPGA. After this configuration stage, the FPGA enters user mode and begins to operate with the user design.

During operation, it can be necessary to subsequently access, and in some instances modify, the configuration registers of the I/O ring. This action is required to enable or disable channels, check status of a block, or to reconfigure to a new data rate or data format. Examples include:

- Memory controller training
- SerDes or PCS link-up
- PCIe enumeration
- Bar support

To monitor the status and modify the configuration, the user design must have access to the subsystem configuration registers.

Within the Speedster device, the configuration and status registers (CSR) form part of the global address map. This map is detailed in "Chapter - 6: Speedster7t NoC Address Mapping" of the Speedster7t Network on Chip User Guide (UG089), and is broken down into particular memory spaces. These spaces can be accessed through a number of mechanisms. This application note describes how to access the global address map using the Speedster FPGA JTAG port, via the ACE Tcl console.

Memory Map

Each device has its own global memory map. The following table details the memory map for Speedster AC7t1500 and AC7t1550 devices. The NoC uses the most significant bits (MSB) in the address to identify the destination space of a transaction. As can be see in the table below, there are several memory spaces, each performing a different function.

Table 1: ***AC7t1500 and AC7t1550 Global Address Map***

Address Bit	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	…	0
Destination																						
PCIe	1	ID	Memory Address																			
DDR4	0	1	Memory Address																			
GDDR6	0	0	0	0	0	Ctrl ID				Memory Address												
NAP	0	0	0	1	0	0	0	NAP Column				NAP Row			Memory Address							
CSR Space	0	0	1	0	0	0	0	0	Target ID						IP ID				Memory Address			
FCU	0	0	1	1	0	0	0	0	0	0	0	0	FCU Address									

Programming Mechanisms

The Speedster device can be programmed using either a .hex file or a .jam file. However, in order to use the Tcl API (see page 6), the device must be programned using a .hex file.

Note

When designing a board with the Achronix Speedster FPGA, include an LED to monitor the `FCU_CONFIG_USER_MODE` signal to ensure that the bitstream is properly downloaded to the device.

Programming Using a .hex File

Speedster FPGA devices can be programmed using a .hex file by following these steps:

1. Enter the Programming and Debug perspective in ACE using either the Programming and Debug Perspective toolbar icon as shown in the figure below or by selecting **Window** → **Open Perspective** → **Programming and Debug**.

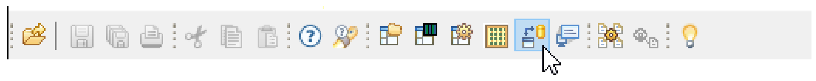
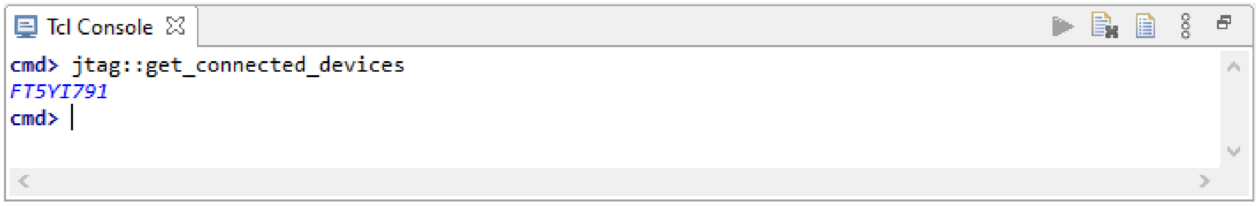


Figure 1: ***The Programming and Debug Perspective Toolbar Icon***

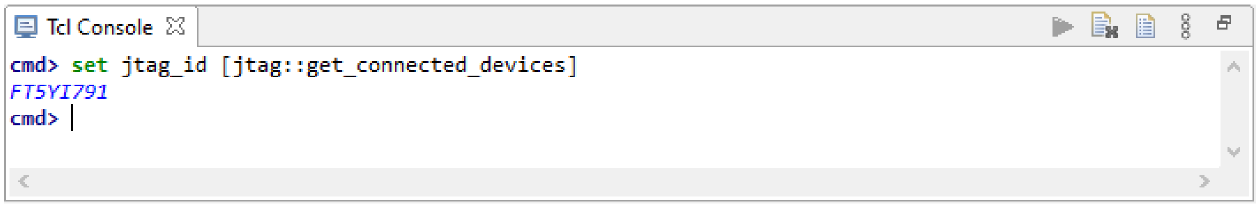
2. In the Tcl Console, enter the command `jtag::get_connected_devices`. The connected Speedster device ID is returned if the device is properly connected to the system.



```
cmd> jtag::get_connected_devices
FT5YI791
cmd>
```

Figure 2: ***Verifying the Connected Device***

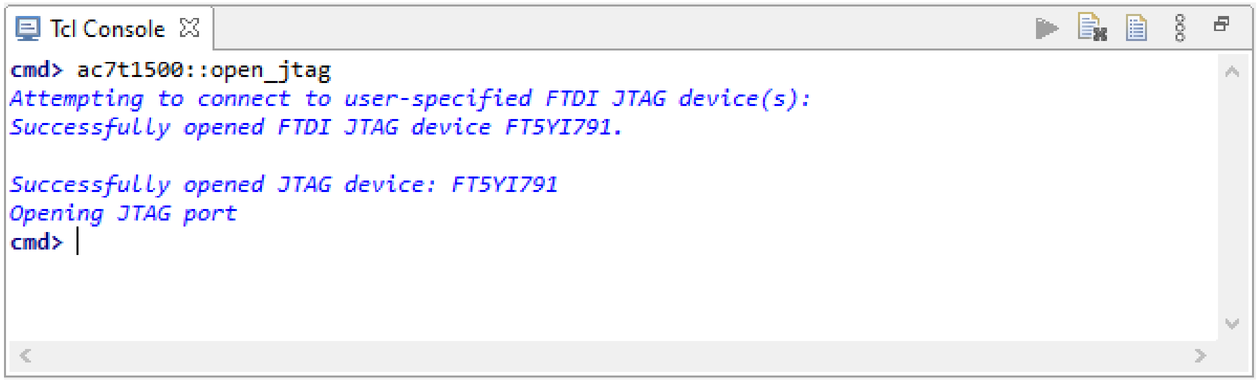
3. After verifying that the device is connected to the system, set the variable `jtag_id` to the device ID returned by the `jtag::get_connected_devices` command as shown.



```
cmd> set jtag_id [jtag::get_connected_devices]
FT5YI791
cmd>
```

Figure 3: ***Setting the `jtag_id` Variable to the ID of the Connected Device***

4. (Optional) Open a JTAG connection with the device using the `ac7t1500::open_jtag` command.

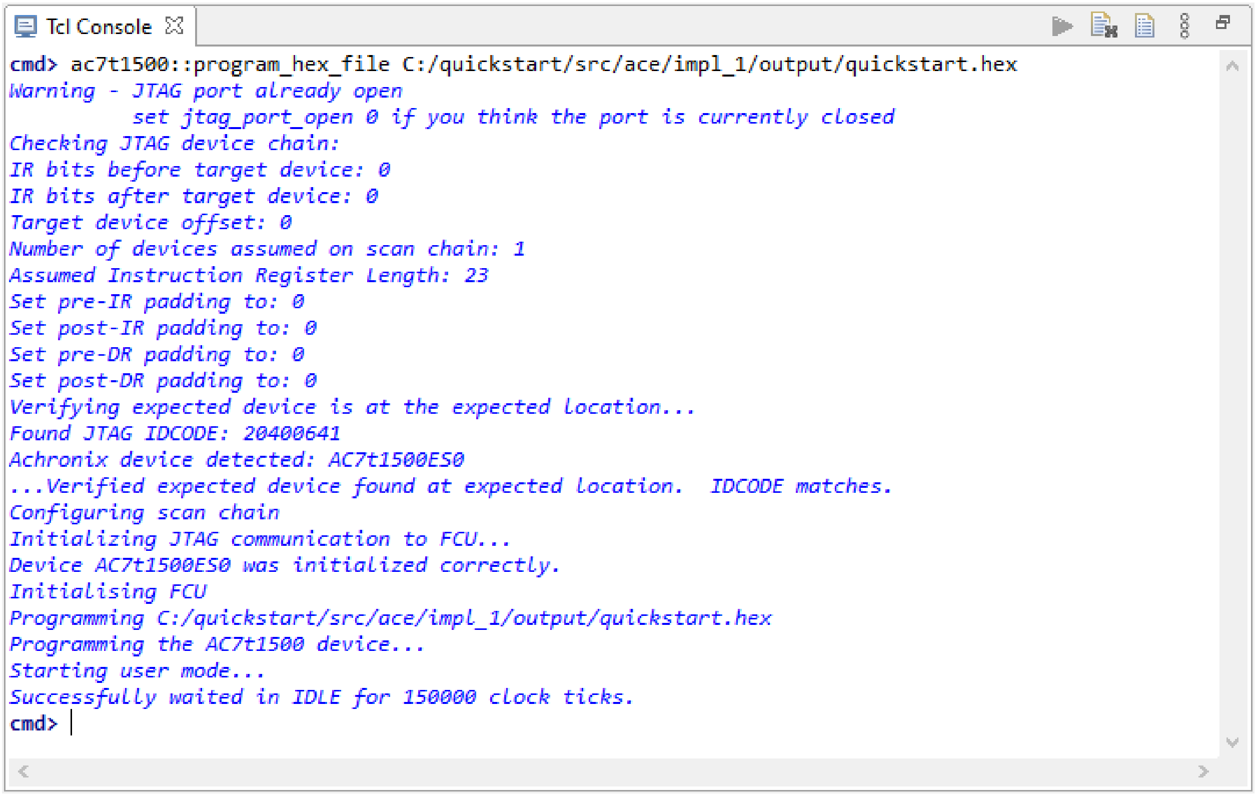


```
cmd> ac7t1500::open_jtag
Attempting to connect to user-specified FTDI JTAG device(s):
Successfully opened FTDI JTAG device FT5YI791.

Successfully opened JTAG device: FT5YI791
Opening JTAG port
cmd>
```

Figure 4: ***Opening a JTAG Connection***

5. Download the .hex file to the device using the `ac7t1500::program_hex_file <path to hex file>` command. This command also automatically runs the `open_jtag` command if the port is not already open making the preceding step optional.



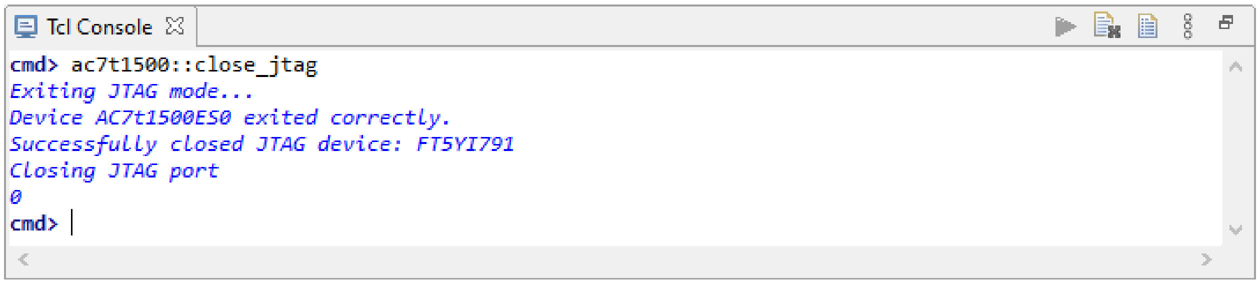
```
cmd> ac7t1500::program_hex_file C:/quickstart/src/ace/impl_1/output/quickstart.hex
Warning - JTAG port already open
          set jtag_port_open 0 if you think the port is currently closed
Checking JTAG device chain:
IR bits before target device: 0
IR bits after target device: 0
Target device offset: 0
Number of devices assumed on scan chain: 1
Assumed Instruction Register Length: 23
Set pre-IR padding to: 0
Set post-IR padding to: 0
Set pre-DR padding to: 0
Set post-DR padding to: 0
Verifying expected device is at the expected location...
Found JTAG IDCODE: 20400641
Achronix device detected: AC7t1500ES0
...Verified expected device found at expected location.  IDCODE matches.
Configuring scan chain
Initializing JTAG communication to FCU...
Device AC7t1500ES0 was initialized correctly.
Initialising FCU
Programming C:/quickstart/src/ace/impl_1/output/quickstart.hex
Programming the AC7t1500 device...
Starting user mode...
Successfully waited in IDLE for 150000 clock ticks.
cmd>
```

Figure 5: ***Downloading a Bitstream to the AC7t1500ES0 Device***

Note

When a JTAG connection with the device is opened, it remains open until it is closed. It is not required to close and reopen a new connection before downloading a bitstream if the JTAG connection is already open.

6. Close the JTAG connection with the `ac7t1500::close_jtag` command.



```
cmd> ac7t1500::close_jtag
Exiting JTAG mode...
Device AC7t1500ES0 exited correctly.
Successfully closed JTAG device: FT5YI791
Closing JTAG port
0
cmd>
```

Figure 6: ***Closing the JTAG Connection***

Warning

There are two groups of commands that can be used to open and close the JTAG port. These are:

1. - `jtag::open`
   - `jtag::close`
2. - `<device namespace>::open_jtag`
   - `<device namespace>::close_jtag`
   - `<device namespace>::program_hex_file`

The `<device_namespace>` commands use a static variable, `jtag_port_open` to track whether the JTAG port is already open. When the above commands are mixed, for example opening the port with `jtag::open`, and subsequently using `ac7t1500::program_hex_file` to program the device, the `jtag_port_open` variable goes out of sequence. The following error is reported:

Cannot open a new connection to FT5YI791. A JTAG connection to FT5YI791 is already open. Please call jtag::close to free up the connection.
0

To avoid this error, it is recommended that the above command groups are not mixed.

If this error occurs, it can be resolved with either of the following commands:

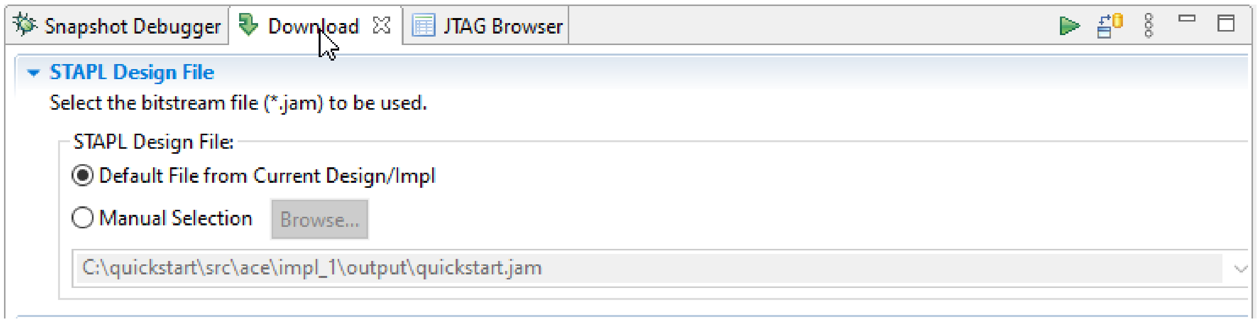
- `jtag::close`
- `set jtag_port_open 1`

The latter command, indicates to the `<device namespace>` commands that the JTAG port is already open.

Programming Using a .jam File

Speedster FPGA devices can be programmed using a .jam file by following these steps:

1. Select the **Download** window in the **Programming and Debug** perspective.

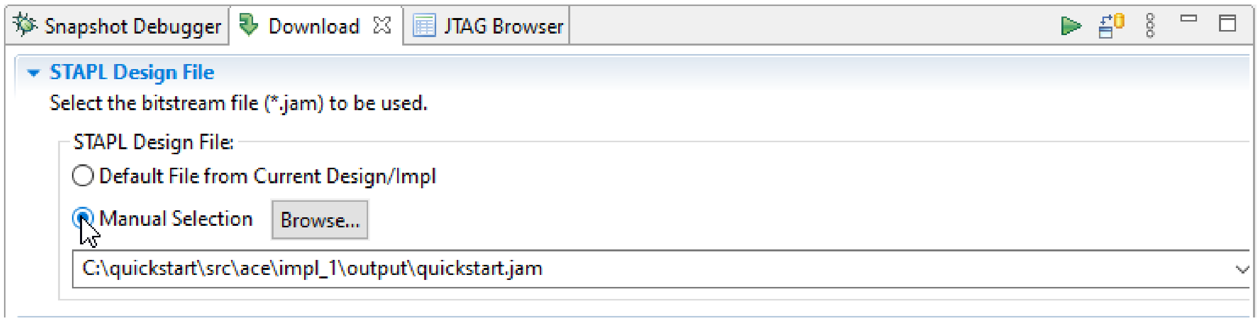


(●) Default File from Current Design/Impl

C:\quickstart\src\ace\impl_1\output\quickstart.jam

Figure 7: *Selecting the Download Window*

2. Use the bitstream from the active implementation (default) or manually select the bitstream by clicking **Manual Selection**.



STAPL Design File

Select the bitstream file (*.jam) to be used.

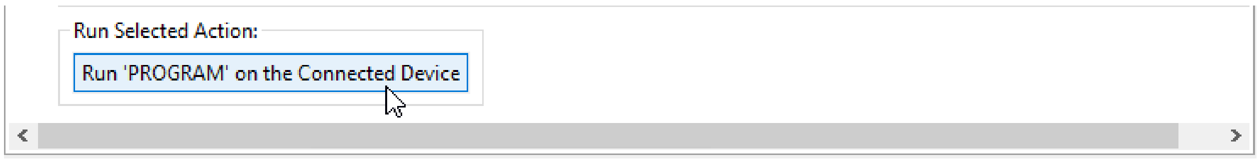
STAPL Design File:

Default File from Current Design/Impl [✓ Manual Selection] Browse...

C:\quickstart\src\ace\impl_1\output\quickstart.jam

Figure 8: ***Manually Selecting a .jam File***

3. Download the bitstream to the device by clicking the **Run 'PROGRAM'** button.



Run Selected Action:

Run 'PROGRAM' on the Connected Device

Figure 9: ***Downloading the Bitstream to the FPGA***

Note

All of the commands above refer to the AC7t1500 device and use the `ac7t1500` namespace. When targeting the AC7t1550 device, it is necessary to change the namespace of the commands to `ac7t1550`. See below for namespace details.

Tcl Register Dictionary

For access via JTAG, using the ACE Tcl console, ACE has a built in API with access mechanisms to each of the device memory areas.

Namespaces

There is a specific Tcl namespace for the API to each device. Therefore each command listed below must be prefixed with its respective device namespace. For example, to read from a named register (see page 12) in the AC7t1500 device, the `ac7t1500::csr_read_named` command must be used and, likewise, to read from an AC7t1550 device named register, the `ac7t1550::csr_read_named` command must be used.

Dictionary Token Hierarchy

The following explains how a very large memory space (2^{42} bits) is broken down into logical areas, which are represented by tokens that can be used to navigate to an individual register.

The AC7t1500 and AC7t1550 use a unified memory space to include the following registers:

- Configuration and status registers (CSR)
- FPGA control unit (FCU) registers
- NoC access point (NAP) registers
- Registers providing access to the GDDR6, DDR4 and PCI memory areas

This 42-bit address can be accessed from many different locations, including from the FCU, the PCIe and from an individual NAP in the design.

The AC7t1500/AC7t1550 memory map, which uses the top address bits to divide the space into the major areas, is detailed below. A function description is provided for each area.

Top Level Memory Map

The following table is based on the memory map (see page 1) above and uses color coding to indicate the hierarchical address spaces.

Table 2: ***Top Level Memory Map Color Coding***

Color	Token Level
🟨	1
🟦	2
🟩	3
🟥	4

Table 3: ***Top Level Memory Map***

Address Bit	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	…	0	Description
Destination																							
PCIe	1	ID	Memory Address																				Allows logic inside the device to access the memory space outside the device via the selected PCIe Controller. Writes into the host controller memory space.
DDR4	0	1	Memory Address																				Provides access to the memory space within the attached external DDR device.
GDDR6	0	0	0	0	0	Ctrl ID			CHID	Memory Address													Provides access to the memory space within the attached external GDDR devices. Some GDDR devices might not be populated.
NAP	0	0	0	1	0	0	0	NAP Column				NAP Row			Memory Address								Provides access to any address map attached to an individual map. The 28-bits of Memory Address are output from the NAP_MASTER to the fabric design attached to that NAP.
CSR Space	0	0	1	0	0	0	0	0	Target ID						IP ID				Memory Address				Provides access to the configuration and status registers. Each part of the device (hard IP, eNoC, PLLs etc) has its own Target ID.
FCU	0	0	1	1	0	0	0	0	0	0	0	0	FCU Address										Provides access to the FCU register space.

Token Hierarchy Levels

The Tcl Test Scripts API uses a Tcl dictionary whereby all locations in the memory map are identified with multilevel text tokens making it easier to find an address without requiring hand computation. The token hierarchy levels are described below.

Level 1

The top level spaces each have their own token i.e., PCIE_SPACE, DDR4_SPACE, GDDR6_SPACE, NAP_SPACE, CSR_SPACE. These spaces, and their associated tokens can be determined using the command `get_dict_spaces` iteratively.

Note

The FCU does not currently have a dictionary space, as the FCU BFM is used for this simulation flow and the FCU BFM does not contain the FCU registers.

Level 2

The available level 2 tokens vary dependent upon the top level space selected. The available tokens are shown in the console outputs below, along with the commands to find these tokens.

Note

Every level 2 area has a "BASE" token. This returns the address at the base (offset = 0) of that address space.

```
# Print to the console the list of memory space keys in the dictionary
set top_level_spaces [get_dict_spaces]

foreach space $top_level_spaces {
    ac7t1500::get_dict_spaces $space
}


## Console output
# Level 1 tokens
Available dictionary spaces at the top level are CSR_SPACE NAP_SPACE PCIE_SPACE DDR4_SPACE
GDDR6_SPACE


# Level 2 tokens per Top level space
Available dictionary spaces for CSR_SPACE are BASE GDDR_0 GDDR_1 GDDR_2 GDDR_3 GDDR_4 GDDR_5
GDDR_6 GDDR_7 CLK_NE CLK_NW CLK_SE CLK_SW GPIO_S GPIO_N DDR4 ETHERNET_0 ETHERNET_1 ENOC_N ENOC_NW
ENOC_NE ENOC_S ENOC_SW ENOC_SE PCIE_0 PCIE_1
Available dictionary spaces for NAP_SPACE are BASE
Available dictionary spaces for PCIE_SPACE are BASE PCIEx8 PCIEx16
Available dictionary spaces for DDR4_SPACE are BASE
Available dictionary spaces for GDDR6_SPACE are BASE GDDR_0 GDDR_1 GDDR_2 GDDR_3 GDDR_4 GDDR_5
GDDR_6 GDDR_7
```

Level 3

Level 3 tokens only exist for those spaces which require an additional level. Only the GDDR6 controllers and the CSR space have a level 3 token:

- The GDDR6 space has tokens CH_0 and CH_1 for each channel within the controller
- The CSR space has a list of IP ID tokens for the selected IP

An example is given below for the Ethernet_0 IP subsystem. The same command can be used to get the IP ID spaces for all the level 2 tokens under the CSR_SPACE

Note

The names of the DBI registers for each PCIe controller can be listed using the `get_dict_spaces` command as shown in the example below. These registers cannot be accessed with any `csr_xxxx` command because they are indirect registers that are read and written via 4 DBI registers that provide the indirection. Instead:

1. Source the `pcie_utils.tcl` file.
2. Use the `pcie_utils::dbi_read` and `pcie_utils::dbi_write` functions to read and write these registers respectively.

```
ac7t1500::get_dict_spaces CSR_SPACE PCIE_0 CORE_REGS
```

```
# Command to get Ethernet_0 IP ID spaces, (level 3)
ac7t1500::get_dict_spaces CSR_SPACE ETHERNET_0


# Console output
Available dictionary spaces for CSR_SPACE ETHERNET_0 are QUAD_PCS_0 QUAD_PCS_1 QUAD_MAC_0
QUAD_MAC_1 400G_PCS_0 400G_MAC_0 400G_MAC_1 CFG_EIU
```

Level 4

Level 4 tokens only apply to the CSR space, and these are the actual register names. An additional feature allows a wildcard to be used on this level to filter the potentially long list of register names returned for certain areas. An example is given below.

```
# Command to get ENOC_NE registers.  Note that there is only one IP ID space, so this defaults to
BASE_IP
ac7t1500::get_dict_spaces CSR_SPACE ENOC_NE BASE_IP


# Console output
The register names for CSR_SPACE ENOC_SW BASE_IP are BASE CL1_CSR_CFG0 CL1_CSR_CFG1 CL2_CSR_CFG
CL3_CSR_CFG CL4_CSR_CFG RIIU_CSR_0_CFG_ACCESS_ENABLE RIIU_CSR_1_CFG_ACCESS_ENABLE
RIIU_CSR_2_CFG_ACCESS_ENABLE RIIU_CSR_3_CFG_ACCESS_ENABLE CLK_RST_TOP_CSR_INOC_31_0
CLK_RST_TOP_CSR_INOC_63_32 CLK_RST_TOP_CSR_INOC_95_64 UNNAMED_0x30 CLK_RST_TOP_CSR_INTERNAL_31_0
CLK_RST_TOP_CSR_INTERNAL_63_32 CLK_RST_TOP_CSR_INTERNAL_95_64 UNNAMED_0x44
CLK_RST_TOP_CSR_INTERNAL_CSR_STATUS CLK_RST_TOP_CSR_USER_MODE
```

Variables under the ACE Tcl Console

When running in the ACE Tcl console, there are two Tcl variables used by the script; one mandatory and one optional. Both can be set in the Tcl console window before running a script:

```
# To set a Tcl variable
set jtag_id ACP1234X
# To clear a Tcl variable
unset jtag_id
# To find out what value a variable is set to
puts $jtag_id
```

Table 4: *Tcl Script Variables*

Variable Name	Mandatory	Description
jtag_id	Yes	Must be set before scripts are run. Must match the JTAG ID value of the particular programming pod. These are the same variables used directly in the JTAG Tcl Commands
quiet_script	No	If set to any value other than 0, calls `jtag::apb_write()` and `jtag::apb_read()` without the -print option. This allows for scripts to run cleanly without excessive console logging.

Dictionary API Commands

Table 5: *Dictionary API Commands*

Command	Arguments	Function	Comments
Interrogate the Dictionaries			
get_dict_spaces	None	Returns the top address map spaces.	PCIE_SPACE, DDR4_SPACE, GDDR6_SPACE, NAP_SPACE, CSR_SPACE.
	<level 1 token>	Returns the level 2 tokens under the top level space.	See list of available tokens under level 2.
	<level 1 token> <level 2 token>	Returns a list of level 3 tokens. If using CSR_SPACE, these are the CSR IP areas under the specific IP.	See level 3 token descriptions.
	CSR_SPACE <IP name> <IP area>	Return a list of CSR registers under the specific IP and IP ID.	Returns register names. Note : Can be a long list.
	CSR_SPACE <IP name> <IP area> <register name>	Get entry for specified register name.	Returns an entry consisting of:`{addr[23:0] reg_size default_value}`
get_csr_reg_name	<address> <value>	Reverse dictionary lookup.	Given the address (must be 11 hex digits), returns the tokens that specify that address. For example, given 08091340264, returns: `get_csr_reg_name() success. The address 08091340264 equates to CSR_SPACE DDR4 PHY MICRORESET`
Named CSR Register Accesses			
csr_write_named	CSR_SPACE <IP name> <IP area> <register name> <value>	Write to the selected register.	Value is treated as hex with or without leading '0x'.

Command	Arguments	Function	Comments
csr_reset_named	CSR_SPACE <IP name> <IP area> <register name>	Reset the selected register to its default value.	Default value is stored in dictionaries.
csr_read_named	CSR_SPACE <IP name> <IP area> <register name>	Read the selected register.	Function returns register read (under ACE).
csr_verify_named	CSR_SPACE <IP name> <IP area> <register name> <value>	Verify the selected register is equal to value.	When run outside of ACE, function always returns 0. The optional argument [expected value] is used in simulation only.
csr_read_all_regs_named	CSR_SPACE <IP name> <IP area>	Read all the registers in an IP area.	Prints register name to console while reading the register.
csr_set_bits_named	CSR_SPACE <IP name> <IP area> <register name> <low_bit> <high_bit>	Set bits [high_bit: low_bit] to 1'b1 in the selected register.	Performs a read-modify-write on the register. To set a single bit, assign `high_bit = low_bit`.
csr_clear_bits_named	CSR_SPACE <IP name> <IP area> <register name> <low_bit> <high_bit>	Clear bits [high_bit: low_bit] to 1'b0 in the selected register.	Does a read-modify-write on the register. To clear a single bit, assign `high_bit = low_bit`.
Based CSR Register Accesses These functions rely on a stateful Tcl flow; the base addresses must first be set before the functions may make calls using the based address. These functions are for scripts focused on a single IP block, and save repeatedly entering the same values.			
csr_named_base	CSR_SPACE <IP name> [IP area]	Set the arguments to be the stateful base address values.	Supports 2 to 3 arguments. If the IP_AREA is not specified, the stateful IP ID variable is set to BASE_IP (=0).
csr_write_based	<register name> <value>	Write to the selected register.	The value is treated as hex with or without leading '0x'.
csr_reset_based	<register name>	Reset the selected register to its default value.	Default value stored in dictionaries.

Command	Arguments	Function	Comments
csr_read_based	<register name>	Read the selected register.	Function returns register read (under ACE).
csr_verify_based	<register name> <value>	Verify the selected register is equal to value.	When run outside of ACE, the function always returns 0.
Individual CSR Access			
csr_named_addr	CSR_SPACE <IP name> <IP area> [register name]	Returns base address of the space.	3 to 4 arguments supported. Address returned is the base address of the arguments provided. If all 4 arguments are provided, address is the full register address.
noc_write	<address> <value>	Write to any location in the address map. (*)	Address must be 42-bit hex (11 characters). Value can be up to 32-bit hex. If `csr_named_addr` is used to obtain the base address, this function can be used by just adding on the offsets to known registers.
noc_read	<address>	Read from any location in the address map.(*)	Address must be 42-bit hex (11 characters). If `csr_named_addr` is used to obtain the base address, this function can be used by just adding on the offsets to known registers.
noc_verify	<address> <value>	Read and verify the result from any location in the address map.(*)	Address must be 42-bit hex (11 characters). If `csr_named_addr` is used to obtain the base address, this function can be used by just adding on the offsets to known registers.
set_bits_addressed	<address> <low_bit> <high_bit>	Set bits [high_bit: low_bit] to 1'b1 in the selected address.	Performs a read-modify-write on the address location. To set a single bit, assign `high_bit` = `low_bit`.
clear_bits_addressed	<address> <low_bit> <high_bit>	Clear bits [high_bit: low_bit] to 1'b0 in the selected address.	Performs a read-modify-write on the address location. To clear a single bit, assign `high_bit` = `low_bit`.
NAP Access These commands access the NAP address space (not the CSR address space).			

Command	Arguments	Function	Comments
nap_axi_write	NAP_SPACE <NAP column> <NAP row> <address> <value>	Create an AXI write from the selected NAP.	Address and data are only 32-bits wide. Address is the AXI write address (awaddr) which does not relate to selecting the NAP, obtained with column and row. Data is relocated to the appropriate byte lane selected by the address, in the 256-bit output from the NAP.
nap_axi_read	NAP_SPACE <NAP column> <NAP row> <address>	Create an AXI read at the selected AXI NAP.	Address and data are only 32-bits wide. Address is the AXI read address (araddr) which does not relate to selecting the NAP, obtained with column and row. Read data is relocated to the appropriate byte lane, selected by the address, in the 256-bit input from the NAP.
nap_axi_verify	NAP_SPACE <NAP column> <NAP row> <address> <value>	Create an AXI read at the selected NAP. Compare the read value against the expected value.	Address and data are only 32-bits wide. Address is the AXI read address (araddr) which does not relate to selecting the NAP, obtained with column and row. Read data is relocated to the appropriate byte lane, selected by the address, in the 256-bit input from the NAP.
GDDR6 and DDR4 Memory Access To write direct into the GDDR memory arrays. To access the GDDR CSR registers, use the CSR commands. The controllers *must* have completed initialization and training for these reads and writes to be successful.			
memory_write	GDDR6_SPACE <controller> <channel> <address> <value>	Write to the selected GDDR memory space	Controller is one of {GDDR_0 to GDDR_7}. Channel is one of {CH_0 CH_1}. Address is up to a 33-bit hex field. Value is up to a 32-bit hex field.
memory_read	GDDR6_SPACE <controller> <channel> <address>	Read from the selected GDDR memory space	Controller is one of {GDDR_0 to GDDR_7}. Channel is one of {CH_0 CH_1}. Address is up to a 33-bit hex field. Returned value is 32-bit hex.
memory_write	DDR4_SPACE <address> <value>	Write to the selected DDR4 memory space	Address is up to a 40-bit hex field Value is up to a 32-bit hex field
memory_read	DDR4_SPACE <address>	Read from the selected DDR4 memory space	Address is up to a 40-bit hex field Returned value is 32-bit hex

Command	Arguments	Function	Comments
Delay or Wait Use this to insert a wait into your command file. When run under ACE, there is approximately 1ms minimum between commands.			
wait_us	<wait value (decimal)>	Add a wait of uS to the simulation command file	Requires a decimal not hex value. As ACE Tcl console commands are measured in milliseconds, only if the value exceeds 1000 (> 1ms) is there an associated ACE command. The wait is always added in simulation.
wait_ns	<wait value (decimal)>	Add a wait of nS to the simulation command file	Requires a decimal not a hex value. Wait is based on the FCU BFM cfg_clk. By default this is 250MHz (4ns). Therefore any delay is in multiples of 4ns. If the cfg_clk frequency is changed, the delays scale accordingly to match the new clock period. This command is only truly applicable to simulation as the time between JTAG commands to the FCU in hardware is on the order of tens of microseconds. As ACE Tcl console commands are measured in milliseconds, only if the value exceeds 1000000 (> 1ms) is there an associated ACE command. The wait is always added in simulation.
Programming Running under ACE only.			
program_hex_file	<hex filename (with extension)> [<optional arguments>]	Programs a hex file. This operation opens the jtag port and leaves it open.	Optional arguments are: -encrypted : Encrypted hex file -do_not_enter_user_mode : Do not enter user mode after programming, stay in configuration mode. This holds most of the IORing IPs in reset. To use Windows backslashes, enclose the filename and path in {} i.e., `program_hex_file {C:\home\me\my_dir\test\my_hex.hex}`

Note

All commands above must be prefixed with the respective device namespace. For example, program_hex_file: `ac7t1500::program_hex_file` or `ac7t1550::program_hex_file`.

(*) Excludes FCU registers. ACE has specific commands to access the FCU registers (see *Speedster7t Configuration User Guide* (UG094) for details). In addition, when used in a simulation flow, these commands require the FCU BFM. FCU registers are only available in simulation when using the FCU RTL.

Caution!

If an invalid read or write is encountered while using the API as shown below, the bitstream must be reprogrammed and the command run again.

The valid bit was never received. Hardware indicates invalid read data at APB address: 0422fff0000

Revision History

Version	Date	Description
1.0	23 Aug 2021	• Initial release.

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Notice of Disclaimer