Z8 Encore!®, eZ80Acclaim!®, and ZNEO™

ZiLOG Nexus Interface API

Reference Manual

RM004502-0506

ZiLOG Worldwide Headquarters • 532 Race Street • San Jose, CA 95126
Telephone: 408.558.8500 • Fax: 408.558.8300 • www.ZiLOG.com

Nexus Interface API
Reference Manual

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126
Telephone: 408.558.8500
Fax: 408.558.8300
www.ZiLOG.com

RM004502-0506

Table of Contents

List of Tables


ZiLOG

Introduction

The Nexus Forum 5001 standard is an open industry standard that provides a general-purpose interface for the software development and debugging of embedded processors.

The Nexus API defines two layers of abstraction: Target Abstraction Layer (TAL) and Hardware Abstraction Layer (HAL). The TAL provides an implementation of the Nexus debug semantics, using the underlying target's on-chip debug hardware. The HAL is the underlying layer that communicates with the target system.

Tools are built on top of the TAL, which is also referred to as the Nexus API. As a general rule, the Nexus API only provides facilities to access the debug hardware features and does not implement functionality that is normally provided in the tool vendor's higher-level APIs. For example, the Nexus API provides facilities to run and single step one instruction. It is up to the tool vendor to provide source level single step, step over, and step out.

In order to allow for processor specific features, the Nexus standard allows vendor-defined extensions in terms of:

- Vendor-defined extensions to standard Nexus operations
- Additional vendor-defined operations
- Additional vendor-defined information in messages

In addition, basic data types associated with addresses, target data and registers are also defined through vendor extensions.

This document defines vendor-specific extensions used in the ZiLOG implementation of the IEEE ISTO-5001 Nexus Standard TAL.

The IEEE ISTO-5001 Nexus Standard is available from the IEEE ISTO organization at http://www.nexus5001.org.

In the context of this document, the term vendor refers to ZiLOG.

Note: Support for Z8 components is not provided in this release. References to Z8 components in Nexus headers are preliminary.

Conventions

The following assumptions and conventions are adopted to provide clarity and ease of use:

Courier Typeface

Commands, code lines and fragments, bits, equations, hexadecimal addresses, and various executable items are distinguished from general text by the use of the `Courier` typeface.

Hexadecimal Values

Hexadecimal values are designated by a lowercase *h* and appear in the `Courier` typeface.

- Example: STAT is set to `F8h`.

Asterisks

An asterisk preceding a parameter denotes the parameter as a pointer.

Trademarks

eZ80®, Z8 Encore!®, and eZ80Acclaim!® are registered trademarks of ZiLOG, Inc. ZNEO™ is a trademark of ZiLOG, Inc.

Online Information

Please visit ZiLOG's website for:

- Product information for Z8 Encore!®, eZ80Acclaim!®, and ZNEO™ microprocessors and microcontrollers.
- Online copies of Z8 Encore!®, eZ80Acclaim!®, and ZNEO™ documentation.

Implementation

The ZiLOG Nexus API extends the standard Nexus API by providing the necessary vendor extensions to support ZiLOG part families, debug communication tools and emulators.

The following ZiLOG debug tools are currently supported by the ZiLOG Nexus API:

- Ethernet Smart Cable
- USB Smart Cable
- ZPAKII
- Serial Smart Cable

Vendor Extensions

The Nexus API uses vendor defined data types, either embedded in standard Nexus data structures or passed directly in to Nexus API functions. Refer to the appendix for a detailed reference of ZiLOG's vendor extension data types.

Naming Conventions

Per the Nexus API specification, the following naming conventions are used for ZiLOG's vendor extensions:

- Data types are prefixed with "nxvt_".
- Constants are prefixed with "NXV_".

Target Address

The type `nxvt_Address`,which specifies a target address, is defined as an unsigned `long` based on the requirement to support addresses up to 32

bits. nxvt_Address is used for setting up events triggered by an address, such as execution breakpoints. nxvt_Address is also used by nx_ReadMem and nx_WriteMem for memory access.

Target Word

The type nxvt_Word, which specifies a target word, is defined as an unsigned long based on the requirement to support 32-bit register values. nxvt_Word is used for setting data-triggered breakpoints and events.

Target Registers

The type nxvt_Registers specifies target register format. Since the format of the registers is processor-dependent, this is defined simply as a buffer to hold the processor's standard register set.

nxvt_Registers is included in the structure that holds received events. The nxvt_Registers type is also used by the control operations NX_CTRL_RESTART_FROM_BREAKSTEP and NXV_CTRL_GET_REGISTERS_ALL.

Target Events

The Nexus API defines event types for break and step. These are supplemented by the event types listed in Table 1. Note that these events are read-only and are therefore prefixed with NXV_READ_EVENT in order to distinguish them from settable events, such as breakpoints.

Table 1. Vendor-Specific Target Event Types

Event Type	Description
NXV_READ_EVENT_HALT_SLP	Target has entered halt or lower power sleep mode.
NXV_READ_EVENT_EXT_RESET	Target reset has been detected.

Table 1. Vendor-Specific Target Event Types

Event Type	Description
NXV_READ_EVENT_POWER_LOSS	Target power loss has been detected
NXV_READ_EVENT_CONNECTION_CLOSED	Target connection has been closed.
NXV_READ_EVENT_BREAK_TRACEFULL	Target has ceased execution due to a trace buffer full condition.
NXV_READ_EVENT_BREAK_EVENT	Target has ceased execution due to an event system match.

Target Spec

The nxvt_VendorDefinedTargetSpec structure contains vendor-specific information required to establish a connection with a target. nxvt_VendorDefinedTargetSpec is embedded within the nxt_TargetSpec structure that is passed to nx_Open.

nxvt_VendorDefinedTargetSpec contains the following members:

- pOpenParameters – String containing target or implementation specific parameters needed during a communication session.
- comminfo – Structure that contains the type of communication method to use between host and target, with associated parameters.

Control Operations

The Nexus API function nx_Control provides general purpose target control. The Nexus specification defines a set of standard control operations while enabling vendors to define processor-specific control operations that extend functionality.

Table 2 lists the ZiLOG Nexus API vendor-specific control operations.

Table 2. Nexus API Vendor-Specific Control Operations

Operation Type	Description
NXV_CTRL_CANCEL	Cancel an operation in progress.
NXV_CTRL_CRC_MEMORY	Calculate CRC over a range of memory.
NXV_CTRL_ERASE_EXT_PROGRAM	Erase external program flash memory.
NXV_CTRL_ERASE_INT_PROGRAM	Erase internal program flash memory.
NXV_CTRL_GET_REGISTER	Get the value of a standard or on-chip peripheral register.
NXV_CTRL_GET_REGISTERS_ALL	Get all standard registers.
NXV_CTRL_GET_TARGET_STATUS	Get the target status (running, halted, etc).
NXV_CTRL_GET_TRACE	Retrieve trace frame.
NXV_CTRL_INITIALIZE_TARGET	Initialize the target debug session.
NXV_CTRL_OTP_CONFIGURE	Configure OTP EPROM
NXV_CTRL_OTP_CONTROL	Perform an OTP control operation (read, burn, blank check, etc.).
NXV_CTRL_SET_EXT_PROGRAM Program external program memory.	Write a value to a standard or on-chip peripheral register.
NXV_CTRL_SET_INT_PROGRAM	Program internal program memory. This may be flash RAM, OTP or other, depending on the processor.

Table 2. Nexus API Vendor-Specific Control Operations

Operation Type	Description
NXV_CTRL_SET_REGISTER	Write a value to a standard or on-chip peripheral register.
NXV_CTRL_SET_ZDI_FREQUENCY	Set the debug frequency.
NXV_CTRL_TRACE_CONTROL	Trace control and configuration.
NXV_CTRL_UPGRADE_FIRMWARE	Upgrade debug tool firmware. Currently supported only for USB Smart Cable and Ethernet Smart Cable.

Control Data

nxvt_VendorDefinedCtrlData is defined as a union of structures, each of which contains the parameters for a given vendor-specific Control Operation. nxvt_VendorDefinedCtrlData is a member of the nxt_CtrlData structure.

The nxt_CtrlData structure is passed in by value to nx_Control. Therefore any data to be returned to the caller must be done through the use of pointer data types within nxvt_VendorDefinedCtrlData.

nxvt_VendorDefinedCtrlData contains the structures listed in the following sections.

setRegister. setRegister contains parameters for NXV_CTRL_SET_REGISTER. It contains the following members:

- type – Indicates whether register is a standard register, on-chip peripheral register, or other.
- address – Register index for standard registers, or address for peripheral register.
- value – The value to write.

getRegister. `getRegister` contains parameters for `NXV_CTRL_GET_REGISTER`. It contains the following members:

- `type` – Indicates whether register is a standard register, on-chip peripheral register, or other.
- `address` – Register index for standard registers, or address for peripheral register.
- `pValue` – Pointer to a long in which to store for the value read.

getRegistersAll. `setRegistersAll` is used with `NXV_CTRL_GET_REGISTERS_ALL`. It contains a pointer to `nxvt_Registers` type to store the standard register values returned.

getTargetStatus. `getTargetStatus` is used with `NXV_CTRL_GET_TARGET_STATUS`. It contains a pointer to an `nxvt_TargetStatusType` to hold the returned target status.

initializeTarget. `initializeTarget` is used with `NXV_CTRL_INITIALIZE_TARGET`. It contains a pointer to a structure containing information about the target configuration that is used to initialize a debug session. The `NXV_CTRL_INITIALIZE_TARGET` command is normally sent immediately after calling `nx_open` in order to establish and initialize the connection to the target.

setIntProgram. `setIntProgram` is used by `NXV_CTRL_SET_INT_PROGRAM`, which is used to program internal flash or OTP memory. It contains the following members:

- `infoPage` – Indicates whether to program the flash info page or normal program memory.
- `address` – The starting address .
- `pData` – Pointer to a buffer containing the data to program.
- `numberOfBytes` – Size of the data in bytes.

eraseIntProgram. `erasetIntProgram` is used by `NXV_CTRL_ERASE_INT_PROGRAM`, which is used to erase internal flash memory. It contains the following members:

- `startPage` – The page to start erase.
- `numberOfPages` – Number of pages to erase.

eraseExtProgram. `eraseExtProgram` is used by `NXV_CTRL_ERASE_EXT_PROGRAM`, which is used to erase external flash memory. It contains the following members:

- `byData` – Indicates whether the erase algorithm sequence consists of data only, or both addresses and data.
- `pBlockAddressList` – List of the first address of each block or sector to erase.
- `numberOfBlocks` – Number of blocks to erase.
- `pDataEraseSeq` – The sequence of bytes to erase a block.
- `pAddrEraseSeq` – Sequence of addresses at which to write erase sequence values (used only if byData is zero).
- `seqSize` – Size of erase sequence.
- `verifyValue` – Status value that indicates erase is complete.
- `verifyMask` – Mask value to AND with status value before comparing to `verifyValue`.

setExtProgram. `setExtProgram` is used by `NXV_CTRL_SET_EXT_PROGRAM`, which is used to program external flash or other type of non-volatile memory. It contains the following members:

- `byData` – Indicates type of programming algorithm to use. If non-zero, the programming sequence consists of writing a sequence of bytes to the address to be written. If zero, the programming algorithm consists of an address/data sequence to unlock the memory, followed by data writes.

- address – Address to start programming.
- pData – The program data.
- numberOfBytes – Size of data, in bytes.
- pWriteSeq – Write sequence (used if byData is nonzero).
- pDataUnlockSeq – Unlock sequence data (used if byData is zero).
- pAddrUnlockSeq – Unlock sequence addresses (used if byData is zero).
- seqSize – Size of write or unlock sequence.
- verifyValue – Status value that indicates erase is complete.
- verifyMask – Mask value to AND with status value before comparing to verifyValue.

upgradeFirmware. upgradeFirmware is used by NXV_CTRL_UPGRADE_FIRMWARE, which is used to update the debug tool firmware. It contains the following members:

- pFirmware – Binary firmware image.
- size – Size of firmware image in bytes.

getTrace. getTrace is used by NXV_CTRL_GET_TRACE, which is used to upload a block of trace frames from the debugger. It contains the following members:

- offset – Offset of first frame to retrieve, relative to the oldest frame in the buffer.
- numberOfFrames – Number of frames to retrieve. May be zero if caller wishes only to query number of available frames.
- pBuffer – Buffer to hold the trace frames. May be NULL if numberOfFrames is zero.
- pAvailable – (optional) Set to total number of frames available upon successful return.

traceControl. `traceControl` is an enumerated type used by `NXV_CTRL_TRACE_CONTROL`, which is used to configure and control trace features. `traceControl` specifies an operation to perform, such as reset trace system, reset trace buffer, and enable break on trace full.

setZdiFrequency. `setZdiFrequency` is used by `NXV_CTRL_SET_ZDI_FREQUENCY`, which is used to set the debugger Frequency for eZ80Acclaim! It contains the following members:

- `frequency` - Desired frequency, in hertz.
- `tableSelect` - Selects which frequency table to use, with zero being the default. Set this to one to use the alternate frequency table.

otpConfigure. otpConfigure is used by `NXV_CTRL_OTP_CONFIGURE`, which is used to set OTP EPROM configuration options. It contains the following members:

- `optionSize` - Size of option data, in bytes.
- `optionData` - The option data.
- `optionMask` - Mask for option data.
- `algoSize` - Size of the algorithm, if included. Algorithm information is optional. Set to zero if `pAlgoData` not set.
- `pAlgoData` - Pointer to a character array containing an algorithm. Algorithm information is optional. Set to zero if no algorithm provided.

otpControl. otpControl contains parameters for `NXV_CTRL_OTP_CONTROL`, which is used to perform OTP operations. It contains the following members:

- `operation` - Indicates the type of OTP operation to be performed. Set to one of `NXV_OTP_CTRL_BLANK_CHECK`, `NXV_OTP_CTRL_WRITE`, `NXV_OTP_CTRL_READ`, `NXV_OTP_CTRL_VERIFY`, or `NXV_OTP_CTRL_READ_OPTIONS`.

- `param` - Operation-specific data. Currently used only with `NXV_OTP_CTRL_READ_OPTIONS` as a pointer to an unsigned long value to store the option data read.

crcMemory. crcMemory contains parameters used by `NXV_CTRL_CRC_MEMORY`.

crcMemory is currently supported only for Z8 Encore! and ZNeo.

- `address` - Address at which to begin CRC calculation. This is used only for ZNEO and must be aligned to 4KB boundary. The address parameter is ignored for Z8 Encore!, which always starts CRC from address zero.
- `numBytes` - Number of bytes to CRC. numBytes is used only for ZNEO, and must e a multiple of 4096. The numBytes parameter is ignored for Z8 Encore!, which performs CRC on entire range of internal flash memory.
- `pCrc` - Pointer to CRC value returned.

Set Event Extensions

The `nxvt_VendorDefinedExtensionSetEvent` and `nxvt_VendorDefinedBasicSetEvent` types are intended to be used to support event features not provided for in the standard Nexus API event types.

`nxvt_VendorDefinedBasicSetEvent` is used to support ZiLOG emulator advanced break and event capabilities that support features beyond simple breakpoints, such as starting and stopping trace on an event trigger. The `nxvt_VendorDefinedBasicSetEvent` structure contains event data which defines the event trigger, a data mask to mask off don't care bits in the event data, and an enumerated type `nxvt_EventAction` that defines the action to take. The event trigger data format is processor-specific. At minimum it consists of status flags, data, and address.

nxvt_VendorDefinedExtensionSetEvent is currently not used, and is simply typedef'ed as an int.

API Commands

This section lists the Nexus API commands implemented by ZiLOG.

void nx_ClearEvent (nxt_Handle * handle, const int eid)

The nx_ClearEvent function clears a debug event previously set with nx_SetEvent.

Only execution breakpoint events are supported by the on-chip debug capability embedded in the processors. The Z8Encore! emulator has richer event capabilities, including support for access and data breakpoints, in addition to execution breakpoints.

The eid value must be a value between breakMinEventId and breakMaxEventId, inclusive. breakMinEventId and breakMaxEventId are members of the nxt_Capability structure in the handle returned from nx_Open. Note that nx_ClearEvent does not return a status, so there is no error feedback if the event ID value is out of range or has no active event associated with it.

Preconditions:

- handle identifies an active Nexus session. It is returned is from a successful invocation of nx_Open.
- eid is the ID of an event previously set up with nx_SetEvent.

Postconditions: Once this function completes, the specified event will be disabled.

Note that target processor execution must be suspended in order to clear breakpoint events.

nxt_Status nx_Close (nxt_Handle * handle)

`nx_Close` terminates an open connection to a target. An attempt to close a connection that is not open will return `NX_ERROR_FAILED`.

Preconditions: Handle is from successful invocation of `nx_Open`.

Postconditions:

- Closes the connection and deallocates the resources allocated for the Nexus session.
- Returned status indicates whether the close was successful.

nxt_Status nx_Control (nxt_Handle * handle, nxt_CtrlData ctrl)

`nx_Control` is used for non-event related target control, such as run, stop, and go. Trace features are also implemented through `nx_Control`.

Preconditions:

- `handle` is from a successful invocation of `nx_Open`.
- `ctrl` specifies the control operation to apply.

Postconditions: If successful, returns `NX_ERROR_NONE`; otherwise may invoke the error callback installed with `nx_Open`, then returns `NX_ERROR_FAILED` or `NX_ERROR_NO_CAPABILITY`

The following control operations are currently supported:

- `NX_CTRL_RESETORHALT` – Put processor in reset, resume execution from current PC, or stop execution.
- `NX_CTRL_CLIENTBREAK` – Enable or disable breakpoints.
- `NX_CTRL_RESTART_FROM_BREAKSTEP` – Restart from a previous breakpoint.

In addition to the standard Nexus operations listed above, several vendor-specific operations are also supported. Refer to Table 2 for a list of vendor-specific operations.

nxt_Status nx_GetEvent (nxt_Handle * handle, nxt_ReceivedEvent * event, int maxBytes, const int block)

The `nx_GetEvent` function is used to read read a target event. Events are generated when target execution is suspended, such as in single step, or halted due to a due to a breakpoint previously set with `nx_SetEvent`. The received event data for single step and breakpoints includes an `nxvt_Registers` structure that contains the target's register state at the time of the event.

Vendor specific events are generated upon detection of processor entering halt mode, external reset, and target power loss. There is no additional data associated with these events.

For eZ80Acclaim! family processors, once the processor enters halt mode it no longer responds to debug commands and therefore requires a reset command to be issued via `nx_Control` to re-enter debug mode. This is not the case for Z8 Encore! and ZNEO family processors.

Preconditions:

- `handle` is a from a successful invocation of `nx_Open`.
- `event` is a pointer to a block of memory to receive the event.
- `maxBytes` is set by the caller to indicate the number of bytes that can be received in the event block of memory.
- `block` specifies whether this function will block until an event is available from the target.

Postconditions: If no event is available, `NX_ERROR_FAILED` is returned. If an event is available, the event data is written to the event buffer passed in, and `NX_ERROR_NONE` is returned.

If the connection to the target is closed while a caller is blocked innx_GetEvent, nx_GetEvent will return event `NXV_READ_EVENT_CONNECTION_CLOSED`.

The following receive event types are supported. Those with the NXV_ prefix are ZiLOG vendor extensions.

- NX_BREAK_STEP
- NXV_READ_EVENT_HALT_SLP
- NXV_READ_EVENT_EXT_RESET
- NXV_READ_EVENT_POWER_LOSS
- NXV_READ_EVENT_CONNECTION_CLOSED
- NXV_READ_EVENT_TRACEFULL
- NXV_READ_EVENT_BREAK_EVENT

nxt_Status nx_GetLastError (nxt_Handle * handle, char * lastError, int maxBytes)

The nx_GetLastError function returns error/status code from most recent nexus API call, and optionally a message containing details for the error. This API was not included in the nx_api.h from IEEE ISTO but is mentioned in the Nexus Standard as an optional feature that can be used as an alternative to the error callback mechanism.

Preconditions:

- handle is a from a successful invocation of nx_Open.
- lastError event is a pointer to a block of memory to receive message. May be set to NULL if caller is only interested in the actual status code.
- maxBytes is set by the caller to indicate the number of bytes that can be received in the block of memory.

Postconditions: Returns Nexus error/status code from last Nexus API call.

nxt_Handle* nx_Open (const nxt_TargetSpec * tSpec, void(* errorCallback)(const char *), nxt_Status * status)

The `nx_Open` function opens a Nexus debug session.

Preconditions:

- `tSpec` specifies the target setup (byte ordering and port type). It also contains the vendor defined `nxvt_VenderDefinedTargetSpec` which contains parameters necessary to establish communication between the host and the target.
- `errorCallback` is a callback function which may be invoked when errors occur. It may be set to `NULL` to prevent callbacks.

Postconditions: If successful, a `handle` is returned and `status` is set to `NX_ERROR_NONE`. If fails, `NULL` is returned, and `status` is set to `NX_ERROR_NO_CAPABILITY`.

The `nxt_Handle` structure contains an `nxt_Capability` structure. The `nxt_Capability` structure is filled out according to the processor's capabilities upon successful open. The `nxt_Capability` structure contains number of breakpoints and memory spaces supported, the Nexus API version and build version, and the debugger firmware version. The `nxt_Handle` type also contains a copy of `tSpec`.

nxt_Status nx_ReadMem (nxt_Handle * handle, const int map, const int accessPriority, const nxvt_Address addr, const size_t numBytes, const int accessSize, void ** bytesRead)

The `nx_ReadMem` function reads from target memory.

Preconditions:

- `handle` from a successful invocation of `nx_Open`.
- `map` specifies a memory map or space (not used for ZNEO).
- `accessPriority` specifies bus priority.

- address is the address to read from.
- numBytes is used to specify how many bytes to read.
- accessSize is used to specify the byte access size to be used during the data transfer.
- bytesRead points to where the data should be stored.

Postconditions: If successful, returns NX_ERROR_NONE and bytesRead points to the data read from the target's memory. If fails, may invoke the error callback installed with nx_Open, then returns NX_ERROR_FAILED.

AccessPriority and accessSize are not currently used by the ZiLOG Nexus API and are ignored. The map parameter specifies the address space. A set of defines with the prefix NXV_ADDR_SPACE_ are defined in nxvtypes.h for this parameter.

Note that target execution must be suspended in order to read target memory.

nxt_Status nx_SetEvent (nxt_Handle * handle, const nxt_SetEvent * setEvent)

Set a breakpoints or single-step event.

Preconditions:

- handle is from a successful invocation of nx_Open.
- setEvent defines the event to set. It is used with breakpoint events, but not single step events.

Postconditions: If successful, returns NX_ERROR_NONE else may invoke the error callback installed with nx_Open, then returns NX_ERROR_FAILED or NX_ERROR_NO_CAPABILITY.

The following event types are supported for all ZiLOG processors:

- NX_ETYPE_STEP

- NX_ETYPE_BREAKPOINT with breakpoint.op set to NX_BREAKPOINT_INSTRADDR

The following event types are additionally be supported for emulators:

- NX_ETYPE_BREAKPOINT with breakpoint.op set to NX_BREAKPOINT_DATA_ADDR, NX_BREAKPOINT_DATAVALUE, or NX_BREAKPOINTDATAADDR_AND_DATAVALUE
- NX_ETYPE_BREAKPOINT with breakpoint.op set to NXV_BREAKPOINT_TRACEFULL
- NXV_ETYPE_EVENT

nxt_Status nx_WriteMem (nxt_Handle * handle, const int map, const int accessPriority, const nxvt_Address addr, const size_t numBytes, const int accessSize, const void * bytesToWrite)

Write to target memory.

Preconditions:

- handle from a successful invocation of nx_Open.
- map specifies a memory map or address space (not used for ZNEO).
- accessPriority specifies bus priority (not used).
- addr is the address to write to.
- numBytes is used to specify how many bytes to write.
- accessSize is used to specify the byte access size to be used during the data transfer.
- bytesToWrite is a pointer to the data to write.

Postconditions: If successful, returns NX_ERROR_NONE and writes specifies data to the target's memory. Otherwise may invoke the error callback installed with nx_Open, then returns NX_ERROR_FAILED.

`accessPriority` and `accessSize` are not currently used by the Nexus API and are ignored. The map parameter specifies the address space. A set of defines with the prefix `NXV_ADDR_SPACE_` are defined in `nxv-types.h` for the map parameter.

Appendix

Glossary

TAL – Nexus Target Abstraction Layer (Nexus API)

HAL – Nexus Hardware Abstraction Layer

API – Application Programming Interface

ICE – In-circuit Emulator

DTL – Data Transfer Language. Communication language used between the host PC and a debug tool.

Nexus – IEEE ISTO-5001 Standard

eZ80 – ZiLOG Acclaim! Family processor

eZ8 – ZiLOG Encore! Family processor

ZNEO – ZiLOG ZNEO family processor

Nexus Vendor Extensions Reference

Defines

```
#define NXV_ADDR_SPACE_eZ8_ROM 0
#define NXV_ADDR_SPACE_eZ8_XRAM 1
#define NXV_ADDR_SPACE_eZ8_RDATA 2
#define NXV_ADDR_SPACE_eZ8_EDATA 3
#define NXV_ADDR_SPACE_eZ8_NVDS 4
#define NX_ADDR_SPACE_eZ8_PRAM 5
#define NXV_ADDR_SPACE_eZ80_RAM 0
```

```
#define NXV_ADDR_SPACE_eZ80_ROM 1
#define NXV_ADDR_SPACE_eZ80_EXTIO 2
#define NXV_ADDR_SPACE_eZ80_INTIO 3
#define NXV_ADDR_SPACE_eZ80_FLASHINFO 4
Note: ZNeo processor has a single unified memory
space, so there are no defined address spaces.

#define NXV_REGISTER_STANDARD (0)
#define NXV_REGISTER_PERIPHERAL (1)

#define NXV_CTRL_SET_REGISTER (0x101)
#define NXV_CTRL_GET_REGISTER (0x102)
#define NXV_CTRL_GET_REGISTERS_ALL (0x103)
#define NXV_CTRL_INITIALIZE_TARGET (0x104)
#define NXV_CTRL_GET_TARGET_STATUS (0x105)
#define NXV_CTRL_ERASE_INT_PROGRAM (0x106)
#define NXV_CTRL_SET_INT_PROGRAM (0x107)
#define NXV_CTRL_ERASE_EXT_PROGRAM (0x108)
#define NXV_CTRL_SET_EXT_PROGRAM (0x109)
#define NXV_CTRL_UPGRADE_FIRMWARE (0x10A)
#define NXV_CTRL_GET_TRACE (0x10B)
#define NXV_CTRL_TRACE_CONTROL (0x10C)
#define NXV_CTRL_SET_ZDI_FREQUENCY (0x10D)
#define NXV_CTRL_CANCEL (0x10E)
#define NXV_CTRL_OTP_CONFIGURE (0x11F)
#define NXV_CTRL_OTP_CONTROL (0x110)
#define NXV_CTRL_CRC_MEMORY (0x111)
```

```
#define NXV_READ_EVENT_HALT_SLP 0x100
#define NXV_READ_EVENT_EXT_RESET 0x101
#define NXV_READ_EVENT_POWER_LOSS 0x102
#define NXV_READ_EVENT_CONNECTION_CLOSED 0x103
#define NXV_READ_EVENT_TRACEFULL 0x104
#define NXV_READ_EVENT_BREAK_EVENT 0x105
#define NXV_ETYPE_EVENT 0x100
#define NXV_BREAKPOINT_TRACEFULL 0x100
```

Typedefs

Types used for target addresses and data in Nexus data structures and API:

```
typedef unsigned long nxvt_Address
typedef unsigned long nxvt_Word
```

Enumerations

nxvt_TargetStatus

The `nxvt_TargetStatus` enumeration type represents the target status. It is used for status values returned from `NXV_CTRL_GET_TARGET_STATUS` control command.

```
typedef enum
{
   NXV_TARGET_STATUS_NOT_CONNECTED,
   NXV_TARGET_STATUS_IN_RESET,
   NXV_TARGET_STATUS_RUNNING,
   NXV_TARGET_STATUS_DEBUG,
   NXV_TARGET_STATUS_HALT_SLP,
```

```
    NXV_TARGET_STATUS_OCD_DISABLED,
    NXV_TARGET_STATUS_UNKNOWN
} nxvt_TargetStatus;
```

NXV_TARGET_STATUS_NOT_CONNECTED. Indicates communication between the host PC and debug tool has not been established, or the connection has been closed.

NXV_TARGET_STATUS_IN_RESET. Indicates the target has been reset by an external source. When a reset is detected, a reset command must be issued from Nexus to re-synchronize communication with target. Alternatively, the connection may be closed and re-opened.

NXV_TARGET_STATUS_RUNNING. The target CPU is running.

NXV_TARGET_STATUS_DEBUG. The target CPU execution has been suspended. Debug commands such as reading and writing memory and registers are only allowed in this mode.

NXV_TARGET_STATUS_HALT_SLP. The target CPU is running in low power mode due to execution of a HALT or STOP instruction.

NXV_TARGET_STATUS_OCD_DISABLE. The on-chip debug (OCD) capability has been disabled. OCD capability is disabled by clearing certain option bits in the internal flash memory. Re-enabling the OCD usually requires mass-erase of the internal flash memory, which resets the option bits to ones. Applicable to ZNEO and Z8 Encore! only.

NXV_TARGET_STATUS_UNKNOWN. Indicates the target status cannot be determined. This may be due to communication problem between the host PC and debug tool, or between the debug tool and the target CPU.

nxvt_EventAction

The `nxvt_EventAction` enumeration type specifies the action to take when an event is triggered. They may be ORed together when multiple actions are needed.

nxvt_EventAction values are used with NXV_SET_EVENT to define the action to take when the event is trigger. Only Z8 Encore! emulators support these complex events.

```
typedef enum
{
   NXV_EVENT_ACTION_NONE = 0,
   NXV_EVENT_ACTION_BREAK = 1,
   NXV_EVENT_ACTION_TRACE_START = 2,
   NXV_EVENT_ACTION_TRACE_STOP = 4,
   NXV_EVENT_ACTION_TRACE_WHILE = 8,
   NXV_EVENT_ACTION_EVENT_IN = 0x10,
   NXV_EVENT_ACTION_EVENT_OUT = 0x20,
   NXV_EVENT_ACTION_EVENT_OUT_PULSE = 0x40,
   NXV_EVENT_ACTION_ARM_NEXT_EVENT  = 0x80
} nxvt_EventAction;
```

NXV_EVENT_ACTION_NONE. Indicates that no action is to be taken. This effectively disables an event.

NXV_EVENT_ACTION_BREAK. Suspend target execution when an event is triggered.

NXV_EVENT_ACTION_TRACE_START. Start capturing trace frames when event is triggered.

NXV_EVENT_ACTION_TRACE_STOP. Stop capturing trace frames when event is triggered.

NXV_EVENT_ACTION_TRACE_WHILE. Capture trace frames as long as the event is active.

NXV_EVENT_ACTION_EVENT_IN. Event action is taken only if the input trigger is active when an event match occurs.

NXV_EVENT_ACTION_EVENT_OUT. Generate an output signal when event is triggered. the output signal level is pulsed if NXV_EVENT_ACtION_EVENT_OUT_PULSE is set. Otherwise, the signal level is toggled.

NXV_EVENT_ACTION_EVENT_OUT_PULSE. Pulse the output when event is triggered. Applicable only if `NXV_EVENT_ACTION_EVENT_OUT` is also set.

NXV_EVENT_ARM_NEXT_EVENT. Arm the next event when event is triggered. the next event is the identified by the current event ID plus one. For example, if event 2 is triggered, then event 3 would be armed if this action is set for event 2.

nxvt_CommType

The `nxvt_CommType` enumeration type specifies the communication protocols used between host PC and target debug tool:

```
typedef enum
{
NXV_COMM_TYPE_ETHERNET = 0,
NXV_COMM_TYPE_SERIAL = 1,
NXV_COMM_TYPE_PARALLEL = 2,
NXV_COMM_TYPE_USB = 3
} nxvt_CommType;
```

NXV_COM_TYPE_ETHERNET. Host PC is connected to debug tool through a TCP/IP Ethernet connection. This type of communication is used with the ZiLOG ZPAKII and Ethernet Smart Cable.

NXV_COM_TYPE_SERIAL. Host PC is connected to a debug tool through the serial port.

NXV_COM_TYPE_PARALLEL. Host PC is connected via parallel port. Currently not supported by any ZiLOG debug tools.

NXV_COM_TYPE_USB. Host PC is connected to a debug tool on USB. This is used with the ZiLOG USB Smart Cable.

nxvt_TargetRegisterType

The `nxvt_TargetRegisterType` enumeration type specifies the type of register to read or write when used with the `NXV_CTRL_GET_REGISTER` and `NXV_CTRL_SET_REGISTER` commands.

```
typedef enum
{
NXV_REGISTER_STANDARD,     NXV_REGISTER_PERIPHERAL,
} nxvt_TargetRegisterType;
```

NXV_REGISTER_STANDARD. Standard registers are internal processor registers, and include the program counter, stack pointer, flags, and general purpose data registers.

NXV_REGISTER_PERIPHERAL. Peripheral registers are used to control behavior of on-chip peripherals, such as UARTs, GPIO Ports, Timers, etc.

nxvt_TraceControl

The `nxvt_TraceControl` enumeration type is used with the `NXV_CTRL_TRACE_CONTROL` operation to control the trace system. Only Z8 Encore! emulators currently support trace.

```
typedef enum
{
NXV_TRACE_CTRL_RESET_BUFFER,
NXV_TRACE_CTRL_RESET_SYSTEM,
NXV_TRACE_CTRL_ENABLE,
NXV_TRACE_CTRL_DISABLE,
NXV_TRACE_CTRL_ENABLE_BREAKONLIMIT,
NXV_TRACE_CTRL_DISABLE_BREAKONLIMIT
} nxvt_TraceControl;
```

NXV_TRACE_CTRL_RESET_BUFFER. Clear the trace buffer to zeroes and reset the trace write pointer to the beginning of the buffer.

NXV_TRACE_CTRL_RESET_SYSTEM. Reset trace system

NXV_TRACE_CTRL_ENABLE. Enable trace capture.

NXV_TRACE_CTRL_DISABLE. Disable trace capture. Note that for Z8 Encore! emulators, the trace system must be disabled in order to use complex events.

NXV_TRACE_CTRL_ENABLE_BREAKONLIMIT. Suspend target execution when trace buffer is full, or the number of frames captured reaches a predefined limit.

NXV_TRACE_CTRL_DISABLE_BREAKONLIMIT. Do not suspect target execution when trace buffer fills or reaches limit.

nxvt_OtpControl

The `nxvt_OtpControl` enumeration type is used with the `NXV_CTRL_OTP_CONTROL` operation to execute an OTP operation.

```
typedef enum
{
NXV_OTP_CTRL_BLANK_CHECK,
NXV_OTP_CTRL_READ,
NXV_OTP_CTRL_WRITE,
NXV_OTP_CTRL_VERIFY,
NXV_OTP_CTRL_READ_OPTIONS,
} nxvt_OtpControl;
```

NXV_OTP_CTRL_BLANK_CHECK. Determine if OTP is blank

NXV_OTP_CTRL_READ. Reads contents of OTP into emulator or OTP Programmer RAM.

NXV_OTP_CTRL_WRITE. Writes contents of emulator or OTP Programmer RAM to OTP

NXV_OTP_CTRL_VERIFY. Verifies contents of OTP against contents of emulator or OTP Programmer RAM.

NXV_OTP_CTRL_READ_OPTIONS. Read current OTP option settings

Structures

nxvt_CommInfo

The `nxvt_CommInfo` structure contains information needed to establish a connection between the host and target. See `nxvt_VendorDefinedTargetSpec` and `nxvt_TargetSpec`.

```
typedef struct {
   nxvt_CommType type;
   union {
```

```
        struct {
           unsigned char flowcontrol;
           unsigned char port;
           unsigned char parity;
           unsigned int baudrate;
           unsigned char databits;
           unsigned char stopbits;

        } serial;
        struct {
           char addr [16];
           int port;
        } ethernet;
         struct {
            char usbId [255];
         } usb;
    } u;
} nxvt_CommInfo;
```

Members

type. Specifies the type of communicaton protocol to use. Valid values `NXV_COMM_TYPE_SERIAL`, `NXV_COMM_TYPE_ETHERNET`, or `NXV_COMM_TYPE_SERIAL`

serial.flowcontrol. Non-zero value enables flow control for serial connection.

serial.port. COM port to user for serial connections. e.g., if using COM1 set to 1, for COM2 set to 2, etc.

serial.parity. Non-zero value enables parity checking for serial connection.

serial.baudrate. Bit rate to use for serial port. Typical values are 19200, 38400, 57600, and 115200.

serial.databits. Number of data bits per character for serial connection, usually set to 8.

serial.stopbits. Number of stop bits per character for serial connection, usually 1.

ethernet.addr. Null-terminated string containing the IP address to used for establishing an Ethernet connection, such as "168.192.1.50"

ethernet.port. TCP port number used with an ethernet connection.

usb.usbId. Vendor or implementation-specific string than uniquely identifies the usb device. If using ZiLOG USB Smart Cable driver, this is set to the USB Smart Cable serial number, which is embedded in the device descriptor. Sample code provided with the Nexus SDK shows how to get the serial number from a connected device.

nxvt_ExtProgramErase

The `nxvt_ExtProgramErase` structure contains external program memory erase parameters to use with `NXV_CTRL_ERASE_EXT_PROGRAM` control command.

```
typedef struct {
   int byData;
   const unsigned long * pBlockAddressList;
   unsigned numberOfBlocks;
   const unsigned char * pDataEraseSeq;
   const unsigned long * pAddrEraseSeq;
   int seqSize;
   int verifyValue;
   int verifyMask;
} nxvt_ExtProgramErase;
```

Detailed Description

Parameters to use with `NXV_CTRL_ERASE_EXT_PROGRAM` control command (eZ80 only).

if `byData` is true, the erase algorithm consists of writing a certain data sequence to the first address of each block to be erased, such as used with Micron MT series Q-Flash parts.

If `byData` is false, then the erase algorithm consists of writing specific data values to specific addresses in sequence. For example, a sequence may consist of writing `0xaa` to address `0x555`, `0x55` to address `0x2AA`, etc., terminated by a write of the block or sector address to be erased. This type of algorithm is used for AMD, Atmel, and STMicro flash memory devices.

For example, to erase the first two sectors of AMD 29LV008B flash located at base address `0x100000`, the parameters should be set as follows:

```
byData = false
pBlockAddressList = { 0x100000, 0x104000 }
numberOfBlocks = 2
pDataEraseSeq =  { 0xAA, 0x55, 0x80, 0xAA, 0x55,
0x30 }
pAddrEraseSeq = {0x555,0x2AA,0x555,0x555,0x2AA, 0}
seqSize = 6
verifyValue = 0x80
verifyMask = 0x80
```

To erase two sectors of Micron MT28F008B located at address `0x100000`, set parameters as follows:

```
byData = true
pBlockAddressList = { 0x100000, 0x104000 }
numberOfBlocks = 2
pDataEraseSeq = { 0x50, 0x20, 0xD0 }
seqSize = 3
verifyValue = 0x80
verifyMask = 0xF8
```

pAddrEraseSeq is not used in this case.

Members

byData. Flag to indicate type of erase algorithm. A nonzero value indicates by-data erase algorithm.

pBlockAddressList. Array containing the start address of each block (or sector) to erase.

numberOfBlocks. Number of blocks to erase.

pDataEraseSeq. Array of bytes that define the erase sequence data.

pAddrEraseSeq . Array of addresses to write the erase sequence data to (used only if !byData).

eqSize. Size of erase sequence.

verifyValue. Status value that indicates erasure is complete.

verifyMask. Mask value to AND with status value to mask off don't care bits before comparing against verifyValue.

nxvt_ExtProgramSet

The `nxvt_ExtProgramSet` structure contains external Program Memory programming parameters to use with `NXV_CTRL_SET_EXT_PROGRAM`.

```
typedef struct {
   int byData;
   unsigned long address;
   const unsigned char * pData;
   unsigned numberOfBytes;
   const unsigned char * pWriteSeq;
   const unsigned char * pDataUnlockSeq;
   const unsigned long * pAddrUnlockSeq;
   int seqSize;
   int verifyValue;
   int verifyMask;

} nxvt_ExtProgramSet;
```

Detailed Description

if `byData` is true, the programming algorithm consists of writing one or more setup byte values to the address to be programmed, followed by the data.

If `byData` is false, the programming algorithm consists of unlocking the flash part by a sequence of writes of specific data values to specific addresses, followed by one or more data writes.

For example, to program 256 bytes of AMD AM29LV008B flash memory at address `0x100000`, set the structure members as follows:

```
byData = false
address = 0x100000
pData = mydata, where mydata is a buffer containing
256 bytes of data
numberOfBytes = 256
pDataUnlockSeq = {0xAA, 0x55, 0xA0 }
pAddrUnlockSeq = { 0x555, 0x2AA, 0x555 }
seqSize = 3
verifyMask = 0xff
```

`verifyValue` is not used here – the verify value is the value written.

To program 256 bytes of Micron MT28F008B flash memory at address `0x100000`, set the structure members as follows:

```
byData = true
address = 0x100000
pData = mydata, where mydata is a buffer containing
256 bytes of data
numberOfBytes = 256
pWriteSeq  = { 0x50, 0x40 }
seqSize = 2
verifyValue =0x80
verifyMask = 0xF8
 note that pDataUnlockSeq and pAddrUnlockSeq are
not used here
```

Members

byData. Flag that indicates type of programming algorithm to use. Non-zero value indicates by-data programming algorithm.

address. Address at which to start programming.

pData. The data to write to the flash memory.

numberOfBytes. Size of data, in bytes.

pWriteSeq. Array containing the flash write sequence (if byData).

pDataUnlockSeq. Array containing the unlock sequence data (if !byData).

pAddrUnlockSeq. Array of addresses at which to write the unlock sequence data bytes (if !byData).

seqSize. Size of write sequence (if byData) or unlock sequence (if !byData).

verifyValue. Status value that indicates successful completion (if byData).

verifyMask. Mask value to AND with status value to mask off don't care bits before comparing against the `verifyValue`.

nxvt_TargetConfiguration

The `nxvt_TargetConfiguration` structure contains target configuration parameters used to initialize the target for a debug session.

```
typedef struct {
   int size;
   char cpuName [32];
   int internalClock;
   unsigned long systemClockFreq;
   unsigned long targetClockFreq;
   unsigned long internalROMSize;
   union {
      struct {
         unsigned char adl;
         unsigned long pc;
         unsigned long spl;
         unsigned long sps;
         int zdiAlternate;
         unsigned long flashLoaderRAM;
         unsigned long flashLoaderRAMSize;
         struct {
            unsigned char enable;
```

```
      unsigned char enableEMAC;
      unsigned char page;
   } internalRAM;
   struct {
      unsigned char enable;
      unsigned char waitStates;
      unsigned char page;
   } internalFlash;
   struct {
      unsigned char lBound;
      unsigned char uBound;
      unsigned char control;
      unsigned char busMode;
   } chipSelects [4];
} eZ80
 struct {
   int configTarget;
   int dac;
   int pinouts;
   int optBitsSize;
   struct {
      unsigned long address;
      unsigned char value;
   } optBits[128];
} eZ8;
 struct {
   float targetVcc;
   unsigned long otpBitMask;
   unsigned long config;
   int fwImageIndex;
} z8;
 struct {
    int externalBusSize;
   struct {
      unsigned char controlH;
      unsigned char controlL;
      unsigned char port;
```

```
            chipSelects[6];
        } zneo;
    } u;
} nxvt_TargetConfiguration;
```

Detailed Description

This structure is used by the `NXV_CTRL_INITIALIZE_TARGET` control command. The data contained within is required to initialize the target to start a debug session.

Processor specific data members are defined in structures within a union.

The eZ80 structure contains information that is unique to eZ80Acclaim! family processors. Not all data is relevant to all processors. For example, if the processor has no internal Flash, the `internalFlash` specific data can be ignored.

The `iceOptions` structure is used only for Z8 Encore! emulators, and should be zeroed when connecting to "real" Z8 Encore! targets.

Members

size. The size of this structure in bytes.

cpuName. Null terminated string containing the CPU name, such as "eZ80F91".

internalClock. If non-zero, indicates target CPU is using internal clock source.

systemClockFreq. The target CPU system clock frequency, in Hertz.

targetClockFreq. Target CPU oscillator clock frequency, in Hertz. This value is the same value as systemClockFreq unless a PLL or divider is used.

internalROMSize. Size of internal flash or OTP memory, in bytes.

eZ80.adl. Initial value of ADL following a reset

eZ80.pc. Initial value for PC following a reset

eZ80.spl. Initial value for SPL following a reset

eZ80.sps. Initial value for SPS following a reset

eZ80.zdiAlternate. If non-zero, use lower ZDI clock frequencies. Using a the lower frequency settings may be required if the normal ZDI clock frequency settings do not give reliable performance.

eZ80.flashLoaderRAM. Address of a portion of RAM that flash erase and programming commands may use for scratch-pad. For the eZ80F92, this should normally be an address in external RAM due to a silicon bug that precludes use of internal RAM when erasing internal Flash RAM pages. If external RAM is not available, mass erase must be used instead of page erase for internal flash.

eZ80.flashLoaderRAMSize. Size of scratchpad RAM. 4 to 8 kb is recommended.

eZ80.internalRAM.enable. If nonzero, internal RAM is enabled.

eZ80.internalRAM.enableEMAC. If nonzero, EMAC ram is also enabled (if available)

eZ80.internalRAM.page. Page at which to map internal RAM. For example, if the CPU has 8KB of internal RAM and page is set to `0xB7`, internal RAM addresses will range from `0xB7E000` to `0xB7FFFF`.

eZ80.internalFlash.enable. If nonzero, internal flash memory is enabled.

eZ80.internalFlash.waitStates. Number of wait status to use for internal flash memory.

eZ80.internalFlash.page. Page at which to map internal Flash. For example, if the CPU has 128k Internal RAM and page is set to 0, Flash RAM will be mapped to the address range `0x000000` to `0x01FFFF`.

eZ80.chipSelects[n].lBound. The upper 8 address bits of the lower bound of chip select n. n is a value between 0 and 3.

eZ80.chipSelects[n].uBound. The upper 8 address bits of the upper bound of chip select n. for example, if a chip select is used for addresses 0x100000 to 0x1FFFFF, lBound should be set to 0x10, and uBound should be set to 0x1F.

eZ80.chipSelects[n].control. The control value for chip select n. Refer to the CPU Product Specification for detailed information.

eZ80.chipSelects[n].busMode. The bus mode value for chip select n. Refer to the CPU Product Specification for detailed information.

eZ8.configTarget(Z8 Encore! emulator only). If true, indicates the following settings are to be used to configure the emulator following reset.

eZ8.dac (Z8 Encore! emulator only). DAC configuration.

eZ8.pinouts (Z8 Encore! emulator only). Output pin configuration.

eZ8.optBitsSize (Z8 Encore! emulator only). Number of option values

eZ8.optBits[n].address (Z8 Encore! emulator only). Array of option addresses.

eZ8.optBits[n].value (Z8 Encore! emulator only). Array of option values.

zneo.externalBusSize. Size of external data bus. Valid values are 0, 8, and 16. Set to zero if external bus is disabled.

zneo.chipSelects[n].controlH. Upper byte of chip select control register. n is a value between 0 and 5, which identifies which of the 6 chip selects. See the ZNeo CPU product Specification for details.

zneo.chipSelects[n].controlL lower. Byte of chip select control register.

zneo.chipSelects[n].port. Which port is used for chip select. Set to 'A' for portA, 'B' for portB, etc.

nxvt_IntProgramErase

The `nxvt_IntProgramErase` structure contains internal flash memory erase parameters to use with `NXV_CTRL_ERASE_INT_PROGRAM`.

```
typedef struct {
   int startPage;
   int numberOfPages;
} nxvt_IntProgramErase;
```

Detailed Description

Internal flash may be erased by page or mass erased. Set `numberOfPages` to -1 to do mass erase.

The number and size of the flash pages are processor-dependent.

Some eZ80Acclaim! Processors have a flash info page in addition to the normal internal program flash memory. The program flash is in pages 0 to `0x7f`, and the flash info page is page `0x80`.

Members

numberOfPages. Number of pages to erase, or -1 for mass erase

startPage. First page to erase

nxvt_IntProgramSet

The `nxvt_IntProgramSet` structure contains internal Program memory programming parameters to use with `NXV_CTRL_SET_INT_PROGRAM` control command.

```
typedef struct {
   int infoPage;
   unsigned long address;
   const unsigned char * pData;
   unsigned numberOfBytes;
} nxvt_IntProgramSet;
```

Detailed Description

Parameters used with `NXV_CTRL_SET_INT_PROGRAM` control command to write to internal flash program memory.

Members

infoPage. Flag that indicates whether writing to the info page or program flash. if non-zero, program info page (eZ80 only).

address. Address at which to start programming

pData. The data to program into the flash memory.

numberOfBytes. Size of data, in bytes.

nxvt_Registers

The `nxvt_Registers` structure holds target register values. Format and contents of standard registers are processor-specific.

```
typedef struct {
   unsigned long pc;
   unsigned char standard [92];
} nxvt_Registers;
```

Members

pc. Program counter. This is separated out from the other standard registers in order to facilitate run-from-break implementation in the nexus API. It may also be included in the standard register block data as well.

standard. Standard register set, including Stack Pointer, Flags, and general purpose registers. Format is processor specific. See "Standard Registers Access" on page 46.

nxvt_VendorDefinedCtrlData

The `nxvt_VendorDefinedCtrlData` structure holds information for vendor defined control operations (see `nx_Control`).

```
typedef struct {
   union {
      struct {
         nxvt_TargetRegisterType type;
         int address;
         unsigned long value;
      } setRegister ;
      struct {
         nxvt_TargetRegisterType type;
         int address;
         unsigned long * pValue;
```

```
} getRegister;
struct {
   nxvt_Registers * pBuf;
} getRegistersAll;
struct {
   nxvt_TargetStatus * pStatus;
} getTargetStatus;
nxvt_TargetConfiguration initializeTarget;
nxvt_IntProgramErase eraseIntProgram;
nxvt_IntProgramSet setIntProgram ;
nxvt_ExtProgramErase eraseExtProgram;
nxvt_ExtProgramSet setExtProgram;
struct {
   const unsigned char * pFirmware
   unsigned size;
} upgradeFirmware;
struct {
   unsigned offset;
   unsigned numberOfFrames;
   unsigned char *pBuffer;
   unsigned *pAvailable;
} getTrace;
struct {
   nxvt_TraceControl operation;
   unsigned long param;
} traceControl;
struct {
   unsigned long frequency;
   int tableSelect;
} setZdiFrequency;
struct {
   unsigned optionSize;
   unsigned long optionData;
   unsigned long optionMask;
   unsigned algoSize;
   const unsigned char *pAlgoData;
} otpConfigure;
```

```
        struct {
           nxvt_OtpControl operation;
              unsigned long param;
        } otpControl;
        struct {
           nxvt_Address address;
           unsigned long numBytes;
           unsigned long *pCrc;
        } crcMemory;
    } u;
} nxvt_VendorDefinedCtrlData;
```

Detailed Description

This structure consists of a union of structures, each of which contains parameters specific to a control command. In keeping with the Standard Nexus types, the union is named `u`.

Members

setRegister.type. type of register to write to. Valid values are `NXV_REGISTER_STANDARD` or `NXV_REGISTER_PERIPHERAL`. Use with `NXV_CTRL_SET_REGISTER`.

setRegister.address. Identifies which register to write to. Set to register index if standard register, or register address if peripheral register. Use with `NXV_CTRL_SET_REGISTER`.

setRegister.value. The value to write to the register. Use with `NXV_CTRL_SET_REGISTER`.

getRegister.type. Type of register to read. Valid values are `NXV_REGISTER_STANDARD` or `NXV_REGISTER_PERIPHERAL`. Use with `NXV_CTRL_GET_REGISTER`.

getRegister.address. Identifies which register to read. Set to register index if standard register, or register address if peripheral register. Use with `NXV_CTRL_GET_REGISTER`.

getRegister.pValue. Pointer to the register value read. Use with `NXV_CTRL_GET_REGISTER`.

getRegistersAll.pBuf. Pointer to buffer to hold standard registers values. See definition of nxvt_Registers structure. Use with NXV_CTRL_GET_REGISTERS_ALL.

getTargetStatus.pStatus. Pointer to status value. Use with NXV_CTRL_GET_TARGET_STATUS.

initializeTarget. Target specific configuration parameters. See definition of nxvt_TargetConfiguration structure. Use with NXV_CTRL_INITIALIZE_TARGET.

eraseIntProgram. Parameters used for erasing internal flash memory. Use with NXV_CTRL_ERASE_INT_PROGRAM.

setIntProgram. Parameters and data to program internal flash memory. Use with NXV_CTRL_SET_INT_PROGRAM.

eraseExtProgram. Parameters used for erasing external flash memory. Use with NXV_CTRL_ERASE_EXT_PROGRAM.

setExtProgram. Parameters and data to program external flash memory. Use with NXV_CTRL_SET_EXT_PROGRAM.

upgradeFirmware.pFirmware. Pointer to buffer containing the firmware image. Use with NXV_CTRL_UPGRADE_FIRMWARE.

upgradeFirmware.size. Size of firmware image, in bytes. Use with NXV_CTRL_UPGRADE_FIRMWARE.

getTrace.offset (Z8Encore emulator only). Trace frame offset relative to oldest frame in the buffer. Since the trace buffer wraps around, the oldest frame is not necessarily the frame at location zero in the buffer. Use with NXV_CTRL_GET_TRACE.

getTrace.numberOfFrames (Z8Encore emulator only). Number of trace frames to retrieve. Use with NXV_CTRL_GET_TRACE.

getTrace.pBuffer (Z8Encore emulator only). Pointer to buffer to store trace frames. Use with NXV_CTRL_GET_TRACE.

getTrace.pAvailable (Z8Encore emulator only). Number of trace frames available in the trace buffer. Use with NXV_CTRL_GET_TRACE.

traceControl.operation (Z8Encore emulator only). Specifies a trace control operation to perform. Use with NXV_CTRL_TRACE_CONTROL.

traceControl.param (Z8Encore emulator only). Operation-specific parameter. Use with NXV_CTRL_TRACE_CONTROL.

setZdiFrequency.frequency (eZ80Acclaim! only). The target clock frequency upon which the ZDI frequence setting is to be based on. Use with NXV_CTRL_SET_ZDI_FREQUENCY.

setZdiFrequency.tableSelect (eZ80Acclaim! only). Set to one to select alternate frequency settings for ZDI Clock. use with NXV_CTRL_SET_ZDI_FREQUENCY.

otpConfigure.optionSize (Z8 only). Size of OTP option data, in bytes. Use with NXV_CTRL_OTP_CONFIGURE.

otpConfigure.optionData (Z8 only). OTP option data. Use with NXV_CTRL_OTP_CONFIGURE.

otpConfigure.optionMask (Z8 only). OTP option data mask. Use with NXV_CTRL_OTP_CONFIGURE.

otpConfigure.algoSize (Z8 only). Size of OTP programming algorithm. Use with NXV_CTRL_OTP_CONFIGURE.

otpConfigure.pAlgoData (Z8 only). OTP programming algorithm. Use with NXV_CTRL_OTP_CONFIGURE.

otpControl.operation (Z8 only). Specifies an OTP operation to perform. Use with NXV_CTRL_OTP_CONTROL.

otpControl.param (Z8 only). OTP operation-specific parameter. Use with NXV_CTRL_OTP_CONTROL.

crcMemory.address. address of block of memory to calculate CRC on. Used for ZNeo only, and must be aligned to 4KB boundary.

crcMemory.numBytes. number of bytes in block of memory to CRC. Used for Zneo only, and must be multiple of 4096.

crcMemory.pCrc. pointer to returned memory CRC value.

nxvt_VendorDefinedTargetSpec

The `nxvt_VendorDefinedTargetSpec` structure contains vendor-specific parameters needed to establish connection to target. See `nxt_TargetSpec` and `nx_Open`.

```
typedef struct {
   char const * pOpenParameters;
   nxvt_CommInfo comminfo;
} nxvt_VendorDefinedTargetSpec;
```

Detailed Description

`pOpenParameters` contains implementation specific parameters. For the ZiLOG Nexus API implementation, this should be set to the path and name of the algorithm ini file.

Members

comminfo. Communication parameters.

pOpenParameters. String containing processor and/or vendor specific target parameters.

nxvt_VendorDefinedBasicSetEvent

The `nxvt_VendorDefinedBasicSetEvent` structure contains information for setting data breakpoints and events. Events are only supported on emulator targets.

```
typedef struct {
   nxvt_EventAction action;
   unsigned char trigger [8];
   unsigned char mask [8];
   nxvt_Address stepEnd;
} nxvt_VendorDefinedBasicSetEvent;
```

Detailed Description

The action, trigger, and mask members are used for setting complex events. An event is triggered when the processor state matches the trigger

settings. The mask can be used to mask off "don't care" bits. Complex events are currently only supported on Z8 Encore! emulators. The format of data contained in trigger and mask arrays is as follows:

```
Byte 0: register address (MSB)
Byte 1: register address (LSB)
Byte 2: register data
Byte 3: CPU flags
Byte 4: PC (MSB)
Byte 5: PC (LSB)
Byte 6: reserved for future use
Byte 7: reserved for future use
```

Members

action. Action to take upon event detection (`NXVT_ETYPE_EVENT`).

trigger. Array containing the conditions that trigger an event for complex events (`NXVT_ETYPE_EVENT`).

Array containing trigger data mask for complex events (`NXVT_ETYPE_EVENT`).

stepEnd. The end address used for stepto events. `Stepto` events are used to step as long as the PC is between the starting PC value and the `stepEnd` value. (`NXVT_ETYPE_STEPTO`).

Standard Registers Access

Z8 Encore!

Table 3 contains the index value to be used with `NXV_CTRL_GET_REGISTER` for reading standard register values. This is also the format of the register block buffer returned by `NXV_CTRL_GET_REGISTERS_ALL` command. Note that where size is 2 bytes, the most significant byte is first in the buffer returned by `NXV_CTRL_GET_REGISTERS_ALL`. Note that the size is given in bytes,

but in some cases not all bits are used. Also note that flags bits can be accessed individually, or as a register.

Table 3. NXV_CTRL_GET_REGISTER Index Values for Z8 Encore!

Register Names	Index	Size in bytes
PC	0	2
SP	1	2
R0	2	1
R1	3	1
R2	4	1
R3	5	1
R4	6	1
R5	7	1
R6	8	1
R7	9	1
R8	10	1
R9	11	1
R10	12	1
R11	13	1
R12	14	1
R13	15	1
R14	16	1
R15	17	1
FLAGS	18	1
RP	19	1
RR0	20	2

Table 3. NXV_CTRL_GET_REGISTER Index Values for Z8 Encore!

Register Names	Index	Size in bytes
RR2	21	2
RR4	22	2
RR6	23	2
RR8	24	2
RR10	25	2
RR12	26	2
RR14	27	2
Carry Flag	28	1
Zero Flag	29	1
Signed Flag	30	1
Overflow Flag	31	1
Decimal Carry Flag	32	1
Half Carry Flag	33	1
User defined Flag 1	34	1
User defined Flag 2	35	1

eZ80Acclaim!

The following table contains the index value to be used with `NXV_CTRL_GET_REGISTER` for reading standard register values. This is also the format of the register block buffer returned by `NXV_CTRL_GET_REGISTERS_ALL` command. Note that where size is

more than one byte, the least significant byte is first in the register block buffer returned by `NXV_CTRL_GET_REGISTERS_ALL`.

Table 4. NXV_CTRL_GET_REGISTER Index Values for eZ80Acclaim!

Register Names	Index	Size in bytes
A	0	1
F	1	1
BC	2	3
DE	3	3
HL	4	3
A'	5	1
F'	6	1
BC'	7	3
DE'	8	3
HL'	9	3
IX	10	3
IY	11	3
I	12	1 or 2 bytes, depending on processor. See processor documentation for size of Interrupt Page Register
R	13	1
ADL	14	1
MADL	15	1

Table 4. NXV_CTRL_GET_REGISTER Index Values for eZ80Acclaim!

Register Names	Index	Size in bytes
MBASE	16	1
IEF1	17	1
IEF2	18	1
SPL	19	3
SPS	20	2
PC	21	3

ZNEO

The following table contains the index value to be used with `NXV_CTRL_GET_REGISTER` for reading standard register values. This is also the format of the register block buffer returned by `NXV_CTRL_GET_REGISTERS_ALL` command. Note that where size is more than one byte, the least significant byte is first in the register block buffer returned by `NXV_CTRL_GET_REGISTERS_ALL`.

Table 5. NXV_CTRL_GET_REGISTER Index Values for ZNEO

Register Names	Index	Size in bytes
R0	0	4
R1	1	4
R2	2	4
R3	3	4
R4	4	4
R5	5	4
R6	6	4

Table 5. NXV_CTRL_GET_REGISTER Index Values for ZNEO

Register Names	Index	Size in bytes
R7	7	4
R8	8	4
R9	9	4
R10	10	4
R11	11	4
R12	12	4
R13	13	4
R14	14	4
R15	15	4
SP	16	4
PC	17	4
FLAGS	18	1
CARRY FLAG	19	1
ZERO FLAG	20	1
SIGN FLAG	21	1
OVERFLOW FLAG	22	1
BLANK FLAG	23	1
USER 1 FLAG	24	1
USER 2 FLAG	25	1
INTERRUPT ENABLE	26	1
PCOV	27	4

Table 5. NXV_CTRL_GET_REGISTER Index Values for ZNEO

Register Names	Index	Size in bytes
SPOV	28	4
CPUCTL	29	1

A

C

E

G

H

I

M

N

O

S

T

V