



FPGA designer's QuickStart guide

Summary

Guide

GU0101 (v1.0) January 26, 2004

This guide gives an overview of using the DXP-based environment to develop an FPGA design. Once you have read this document, refer to the linked documents for a detailed description of that area of the design process.

Over the last 50 years the electronics engineer has had a rapidly changing palette to work with. The introduction of the transistor in 1947 heralded the arrival of solid-state electronics, fostering the development of binary – or digital electronics. With the implementation of multiple transistors on a single piece of silicon in 1959 the integrated circuit (IC) was born. With it came the application of Boolean logic – a form of algebra where all values are reduced to true or false – giving rise to the computer age.

The spread of computers throughout the developed world, and the rapid improvements in IC development capabilities saw more and more transistors being squeezed onto an IC. The result of this has been more and more powerful devices, identified by the term large scale integration, or LSI circuits. This process has continued in harmony with the introduction of numerous computer interface standards. Bringing together LSI fabrication capabilities with these defined standards has resulted in the development of powerful, application specific integrated circuits (ASICs) for networking, communications, image processing, computer bus management, and so on.

Typically these components are combined with microprocessors and other logic to form sophisticated electronic products, capable of performing an incredible variety of tasks – each solving some problem that the engineer set out to resolve.

Along with the growth in the size and functionality of application-specific ICs, there has been a corresponding growth in the size and capabilities of programmable logic. Larger programmable devices typically have their functionality arranged as an array of general purpose logic blocks, with programmable interconnections between them. These are known as Field Programmable Gate Arrays (FPGAs).

With their ability to operate at high switching frequencies FPGAs have provided an ideal solution for implementing large amounts of high speed signal processing circuitry, allowing the designer to reduce the size and cost of a product.

Today these devices have sufficient capacity to implement more than just some of the hardware in a product – they can potentially be programmed to implement an entire digital system, including the microprocessor, peripheral components and the interface logic.

To do this the engineer needs a design environment that solves the system integration issues – where they can capture the hardware design, write the embedded software for the processor, and implement, test and debug both the hardware and software on the target FPGA.

Altium's FPGA design software brings together the required tools and the necessary communications systems. Combine this with an FPGA implementation platform – the NanoBoard – and you have a complete FPGA design environment. This QuickStart guide will give you an overview of how you can capture and implement an entire digital system in an FPGA in this design environment.

Getting started with FPGA Design

Product documentation

The product documentation is structured as a set of focused tutorials, application notes, guides, articles, reference manuals and online help. The entire set of documentation can be accessed from the help system (**Help » Contents**), where it is presented in either PDF or CHM format. The online help also includes a more detailed description of the various document kinds available, as well as information on how each kind of document presents in the help navigation system.

Examples and Reference Designs

There are a large number of example designs included with the software, in the `C:\Program Files\Altium2004\Examples` folder. FPGA focused examples include:

- **FPGA Core Integration** – simple FPGA project and related core component project.
- **FPGA Design Tips** – projects that demonstrate a feature of the design system, including projects that demonstrate bus constraints and bus interconnects.
- **FPGA Hardware** – designs that do not include an embedded processor.
- **FPGA Processor Tests** – projects for testing the functionality of the NanoBoard.
- **FPGA Processors** – processor-based projects that demonstrate a specific MCU and features on the NanoBoard.
- **FPGA Third Party Boards** – designs that can be implemented in an FPGA on a variety of 3rd party development boards.
- **NanoBoard Testing** – designs used for testing the functionality of the NanoBoard, referred to in the [NanoBoard Technical Reference Manual](#).
- **Reference Designs** – working designs that include both an FPGA project and a PCB project.
- **Tutorials** – files used by the tutorials included in the documentation.

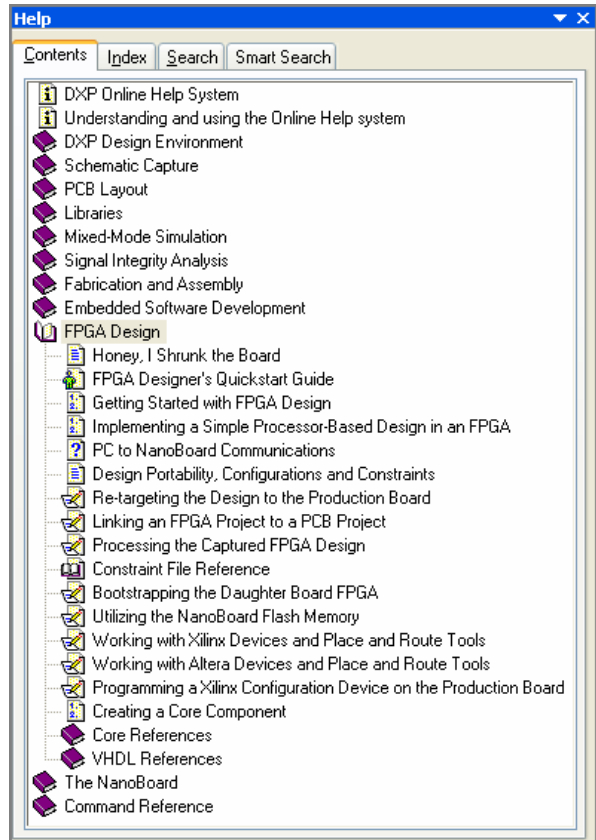


Figure 1. All documentation is available in the on-line help

An Overview of the Design Process

Altium's FPGA design environment allows you to design, implement and debug a microprocessor-based digital design in an FPGA. The design is captured as a schematic, or using a mixture of schematic and VHDL. The embedded software is written in a coding-aware editor, ready for compilation and download onto the processor in your design.

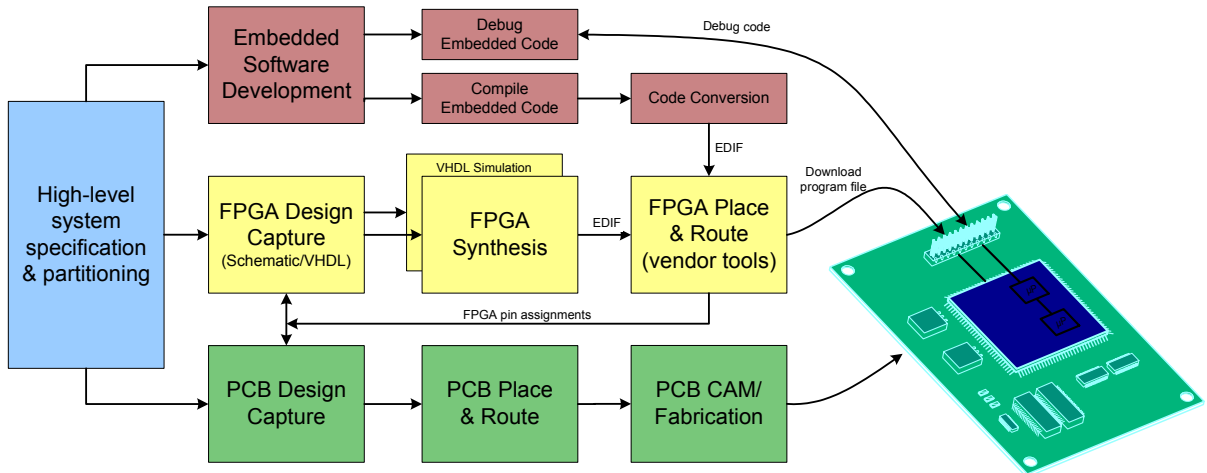


Figure 2. System diagram, showing the flow of the hardware design, embedded software, and PCB design.

Once the hardware design is complete it is synthesized, a process that transforms it from the capture form into a low-level gate form.

After design synthesis a place and route is performed, a process where device-aware software implements the design in the target FPGA. The Vendor-specific place and route software required to synthesize for the target architecture is operated by the DXP environment, which automatically manages all project and file handling aspects required to generate an FPGA program file.

To test and debug the design the system includes a NanoBoard, an implementation platform that includes an FPGA, as well as an array of general purpose peripheral components. The software communicates directly with the NanoBoard via a port on the PC, programming the FPGA and implementing your design.

Once the design has been implemented on the NanoBoard it can be debugged, using virtual instruments and boundary scan pin status technology to debug the hardware, and the integrated debugger for the embedded software. Since debugging is performed live from within the same environment as the design is captured in, design iterations can be carried out quickly and software/hardware solutions rapidly explored.

Flow diagram of the FPGA design process

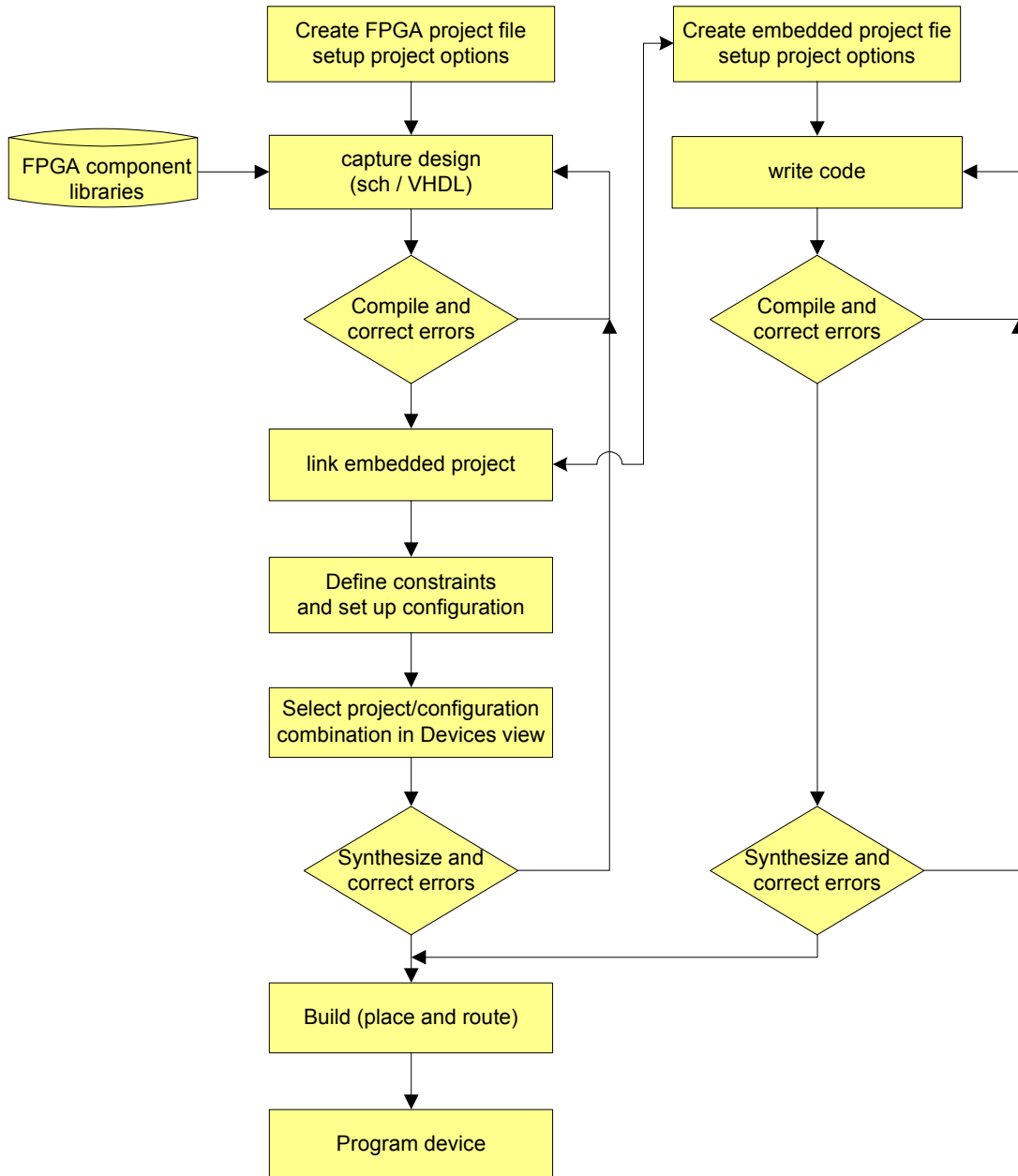


Figure 3. Flow diagram of the design process

Capturing the FPGA project

The basis of every design created in the DXP environment is a project file. Multiple types of design projects are supported, including:

- PCB projects (*.PrjPcb)
- FPGA projects (*.PrjFpg)
- Embedded projects (*.PrjEmb)
- Core projects (*.PrjCor)
- Integrated libraries (*.IntLib)
- Script projects (*.PrjScr)

Most projects targets a single implementation – for example a PCB project becomes one PCB, and an FPGA project is implemented in a single FPGA.

The project document itself is an ASCII file that stores project information, such as the documents that belong to the project, output settings, compilation settings, error checking settings, and so on.

The hardware design in an FPGA project is captured as a set of schematic sheets, VHDL code, or a mixture of both. The schematic is captured in the schematic editor, with each schematic sheet being stored as a separate file. VHDL is captured in the syntax-aware VHDL editor. Click the **Project** button on the **Projects** panel to add new source documents into the project.



For detailed information on how to create an FPGA project, add schematic sheets, place and wire components and implement the design in an FPGA, refer to the tutorial [Getting Started with FPGA Design](#).

Structuring a multi-sheet project

While the project file links the various source documents into a single project, the document-to-document and net connective relationships are defined by information in the documents themselves.

The design is partitioned into logical blocks, each block is represented on the top schematic sheet by a sheet symbol. The Filename attribute of each sheet symbol references the sub-sheet (or VHDL file) that it represents. A sub-sheet can also include sheet symbols referencing lower schematic sheets. Using this approach a design hierarchy of any depth or complexity can be created. Figure 4 shows the hierarchy of a multi-file design after it has been compiled.



For more information on multi-sheet designs, refer to the article [Connectivity and Multi-sheet Design](#).

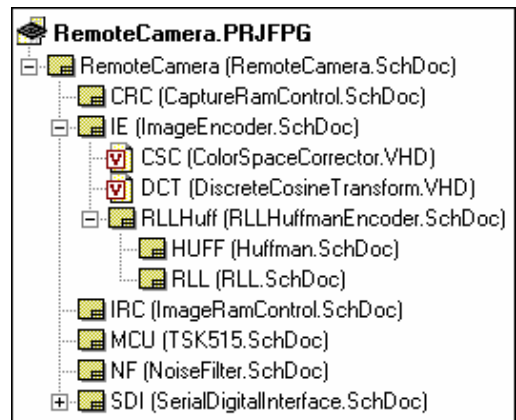


Figure 4. A compiled FPGA project, showing the hierarchical relationship between project documents.

Building and maintaining a hierarchical project

There are a number of commands available to speed the process of building hierarchy in a multi-sheet design. These include:

Create sheet from symbol – use this schematic editor command to create a sheet below the nominated sheet symbol. Matching Ports will be added to the sub-sheet, ready to wire.

Create VHDL file from symbol – use this schematic editor command to create a shell VHDL file, with an entity declared that includes port definitions to match the sheet entries in the nominated symbol.

Create symbol from sheet – use this schematic editor command to create a symbol from the nominated sheet. Make the sheet that is to include the sheet symbol the active document before launching this command.

Create component from sheet – use this schematic editor command create a schematic component symbol from the current sheet, whose pins match the ports on the schematic sheet. Use this when designing a core component, running it will create the schematic symbol that represents the core in a new library.

Create schematic part from file – create a schematic component symbol from the current VHDL file, whose pins match the port definitions declared in the entity. Use this when designing a core component, running it will create the schematic symbol that represents the core in a new library.

Once a multi-sheet design has been created, use the *Synchronize Ports to Sheet Entries* dialog to maintain the sheet symbol to matching sub-sheet connections (select **Synchronize Sheet Entries and Ports** from the **Design** menu).

Implementing repeated sections in a design

One of the advantages of incorporating an FPGA into a design is their ability to implement large amounts of repetitive circuitry. The environment includes features specifically to support projects with repetitive circuitry – as well as the singular *one sheet symbol = one sub-sheet* representation, you can also create a structure where the same sub-sheet is referenced many times.

This is known as multi-channel design. There are 2 approaches to multi-channel design, either by referencing the same sub-sheet from multiple sheet symbols, or using one sheet symbol with the Repeat keyword. When the design is compiled any repeated sections (or channels) are automatically instantiated the required number of times. The advantage of multi-channel design is that it allows you to maintain a single copy of the source channel, even after the design has move to implementation in the FPGA or on the PCB.



For more information on how to capture a design with repeated sections, refer to the article [Multi-Channel Design Concepts](#).

Mixed schematic/VHDL document hierarchy

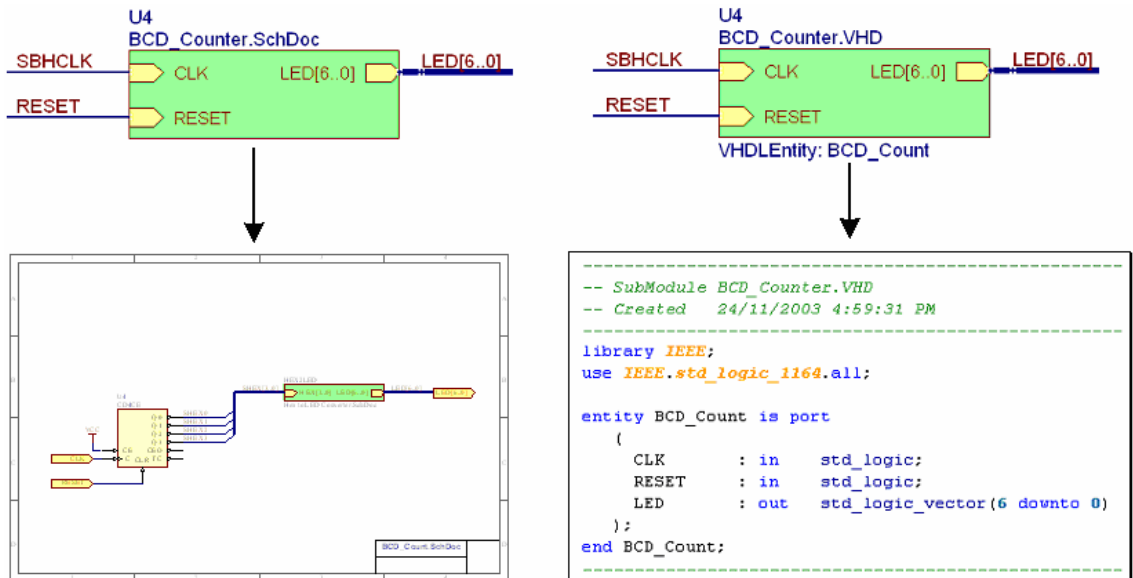


Figure 5. Document hierarchy is created by placing sheet symbols to represent the document below.

VHDL sub-documents are referenced in the same way as schematic sub-sheets, by specifying the sub-document filename in the sheet symbol that represents it. The connectivity is from the sheet symbol to an entity declaration in the VHDL file. To reference an entity with a name that is different from the VHDL filename, include the VHLEntity parameter in the sheet symbol, whose value is the name of the Entity declared in the VHDL file, as shown in Figure 5.

Wiring the Design

Connectivity between the component pins is created either by physical connectivity, or logical connectivity. Physical connectivity is created by placing wires to connect component pins to each other. Logical connectivity is create by placing matching net identifiers, such as net labels, power ports, ports and sheet entries. When the design is compiled the connectivity is established, according to the net identifier scope defined for the project.



Note that while the environment supports compiling projects using either a flat or hierarchical connective structure, FPGA projects must be hierarchical.

Establishing connectivity between documents

Hierarchical net and bus connectivity between documents obeys the standard hierarchical project connection behavior, where ports on the sub-document connect to sheet entries of the same name in the sheet symbol that represents that document, as shown in Figure 6.

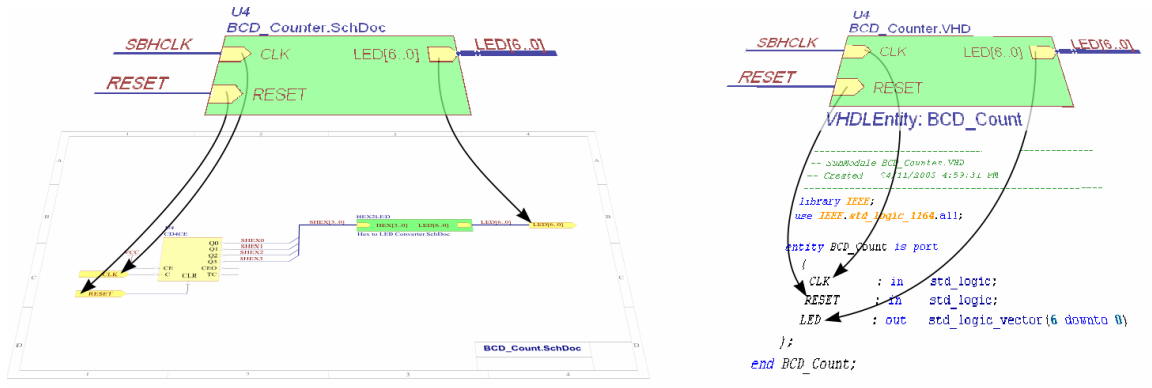




Figure 6. Hierarchical net connectivity is from the sheet entries to matching ports on the document below.

-  For details on placing the wiring, refer to the tutorial [Getting Started with FPGA Design](#).
-  For more information on connectivity in multi-sheet designs, refer to the article [Connectivity and Multi-sheet Design](#).

Using buses and bus joiners

Typically there are a large number of related nets in a digital design. Buses can play an important role in managing these nets, and help present the design in a more readable form.

Buses can be re-ordered, renamed, split, and merged. To manage the mapping of nets in buses, there is a special class of component, known as a bus joiner. Bus joiners can be placed from the FPGA Generic integrated library (bus joiner names all start with the letter J). Figure 7 shows examples of using bus joiners. There are also many examples of using bus joiners in the example designs in the software.

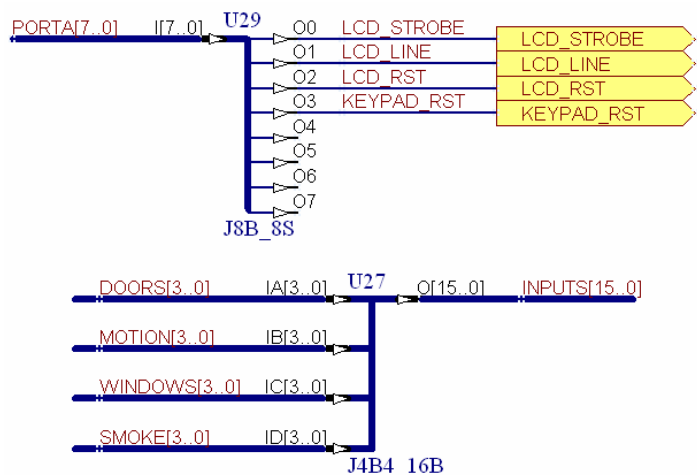


Figure 7. Examples of using bus joiners.



Note that apart from the JB-type joiner, all bus joiner pins have an IO direction – use the correct joiner to maintain the IO flow. Pin IO can be displayed on sheet, enable the **Pin Direction** option in the schematic *Preferences* dialog.

Bus joiner splitting / merging behaviour

The basic rule is that bus joiners separate/merge the bits (or bus slice) from least significant bit (or slice) down to most significant bit (or slice).

For example, in Figure 8, U17 splits the incoming 8-bit bus on pin I[7..0] into two 4-bit bus slices, OA[3..0] and OB[3..0]. Obeying the *least to most* mapping at the slice level, the lower four bits of the input bus map to OA[3..0], and the upper four bits map to OB[3..0]. Following this through to the bit level, I0 will connect to OA0, and I7 will connect to OB3.

The other joiner shown in Figure 8 merges the four incoming 4-bit slices into a 16-bit bus. With this joiner IA0 connects to O0, and ID3 connects to O15.

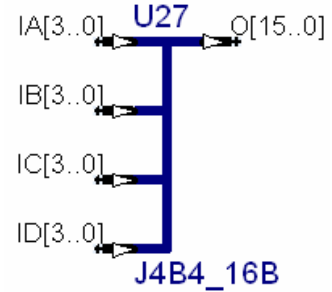
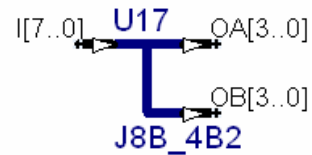


Figure 8. Splitting/merging bus slices

Matching buses of different widths using the JB-type bus joiner

The JB-type bus joiner allows you to match nets in buses of different widths. It does this via 2 component parameters, IndexA and IndexB that map from one bus through to the other bus. These indices must be defined when you use a JB joiner.

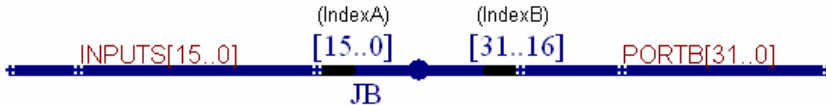


Figure 9. JB-type bus joiner, note that there is no IO direction for a JB component

Read the flow of nets through a JB-type bus joiner by matching from the nets in the attached bus, to the first index on the bus joiner, to the second index in the bus joiner, to the nets defined in the second bus net label.

Left Bus ↔ IndexA ↔ IndexB ↔ Right Bus

The rules for matching nets at each of the ↔ points are as follows:

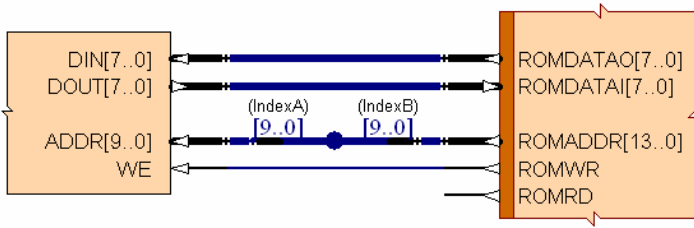


Figure 10. An example of using the JB bus joiner to achieve sub-set mapping.

- If both bus ranges are descending, match by same bus index (one range must lie within the other for valid connections). In Figure 10 the matching is:

ADDR9 ↔ IndexA9 ↔ IndexB9 ↔ ROMADDR9, thru to
 ADDR0 ↔ IndexA0 ↔ IndexB0 ↔ ROMADDR0

(In this example ROMADDR10 thru ROMADDR13 will be unconnected)



Figure 11. Using of a bus joiner for offset mapping.

- In Figure 11 the matching is:

INPUTS15 ↔ IndexA15 ↔ IndexB31 ↔ PORTB31, thru to
 INPUTS0 ↔ IndexA0 ↔ IndexB0 ↔ PORTB16



Figure 12. Using a bus joiner for range inversion.

- If one bus range is descending and another is ascending, the indices are matched from left to right. In Figure 12 the matching is:

INPUTS0 ↔ IndexA15 ↔ IndexB31 ↔ PORTB31, thru to
 INPUTS15 ↔ IndexA0 ↔ IndexB16 ↔ PORTB16



Figure 13. Another example of using a bus joiner for range inversion.

- In Figure 13 the matching is:

INPUTS15 ↔ IndexA15 ↔ IndexB31 ↔ PORTB0, thru to
 INPUTS0 ↔ IndexA0 ↔ IndexB16 ↔ PORTB15



For an example of using bus joiners, refer to the example C:\Program Files\Altium2004\Examples\FPGA Design Tips\Bus Interconnect\Interconnect.PRJFPG.

FPGA-ready schematic components

A wide variety of FPGA-ready schematic components are included with the system, ranging from processors, to peripheral components, down to generic logic. The hardware design is captured by placing and wiring these schematic components, or writing VHDL. The FPGA-ready schematic components are like traditional PCB-ready components, except instead of the symbol being linked to a PCB footprint each is linked to a pre-synthesized EDIF model.

The FPGA-ready component libraries are in the folder
`\Altium\Library\Fpga.`

As well as components that you use to implement your design, the available FPGA libraries include components for the virtual instruments, and the components that are mounted on the NanoBoard and are accessible via the pins on the FPGA. The role of each type of component is described below.

Model linkage

EDIF model linkage is not handled like standard component model linkage, since the model must be chosen to suit the target device. For EDIF models the target device family is used to select the correct folder of EDIF models (for example `\Xilinx\Spartan2E`), and then the component's Library Reference is used to select the EDIF model file from within that folder. Models included with the system are stored in a hierarchy of folders under `\Program Files\Altium2004\Library\Edif`.

As well as system supplied models, user-created pre-synthesized EDIF models are supported. These can be stored in a user model folder, this folder is specified in the *FPGA Preferences* dialog (accessed via the **Tools** menu in the schematic or VHDL editor when the active project is an FPGA or a Core project). User models can also be stored in a hierarchy of folders if you are developing a model for multiple target devices.

The search sequence for EDIF models is:

```
$project_dir
$user_edif$vendor$family
$user_edif$vendor
$user_edif
$system_edif$vendor$family
$system_edif$vendor
$system_edif
```

Pre-synthesized user models are developed by creating a Core project, whose EDIF output becomes the model for your user-defined component. There are a number of features to support this process, including commands to synthesize for all targets, publish the EDIF model (package it with all other required EDIF models), and generate a component symbol to represent the core.



For more details refer to the tutorial, [Creating a Core Component](#). The tutorial also details how to use that component in an FPGA project while still developing the core.



For an example of a core component project that is linked to an FPGA project, open the design **Workspace** `C:\Program Files\Altium2004\Examples\FPGA Core Integration\LCD Controller And Keypad\LCD_Keypad.DSNWRK`. To use this example you must define a user model location first, then generate the model for the keypad scanner before attempting to process the FPGA design (LCD_Keypad) that uses the model.

Components to implement your design

A range of processors, support peripherals and libraries of interface logic are available to implement the hardware in your FPGA design. The exact set of components that are available for FPGA design will depend on the Altium product you are using.

Processor cores and memory

Processors can be placed from the `\Program Files\Altium\Library\Fpga\FPGA Processors.IntLib` library.

The Nexar product supports the following processors (and related embedded software tools):

- TSK165 – Microchip 165x family instruction set compatible MCU
- TSK51 – 8051 instruction set compatible MCU
- TSK80 – Z80 instruction set compatible MCU



The on-line help system includes a hardware reference manual for each processor, complete with instruction set details. Navigate in the help to FPGA design - Core References - Processors.

Peripheral components

Peripherals can be placed from the `\Program Files\Altium\Library\Fpga\FPGA Peripherals.IntLib` library.

CAN Controller – parallel to serial interface, implementing a Controller Area Network serial communications bus on the serial side. The CAN serial bus provides high bit rate, high noise immunity and error detection. The Controller implements the BOSCH CAN 2.0B Data Link Layer Protocol. The CAN controller can be used in conjunction with the CAN interface hardware on the NanoBoard.

FPGA Startup – user-definable power-up delay, used to implement power-on reset. An internal counter starts on power up, counting the number of clock cycles specified by the Delay pin, the output pin being asserted when the count is reached.

I2C – parallel to serial interface, implementing an Inter-Integrated Circuit (I2C) 2-wire serial bus on the serial side. Controllers only support a single master I2C serial bus system. The I2C controller can be used in conjunction with the I2C interface hardware on the NanoBoard.

Keypad Controller – 4 by 4 keypad scanner with de-bounce. Can be used in a polled or interrupt driven system. Available in either Wishbone or non-Wishbone variants. The Keypad controller can be used in conjunction with the keypad on the NanoBoard.

LCD Controller – easy to use controller for a 2 line by 16 character LCD module. The LCD controller can be used in conjunction with the LCD display on the NanoBoard.

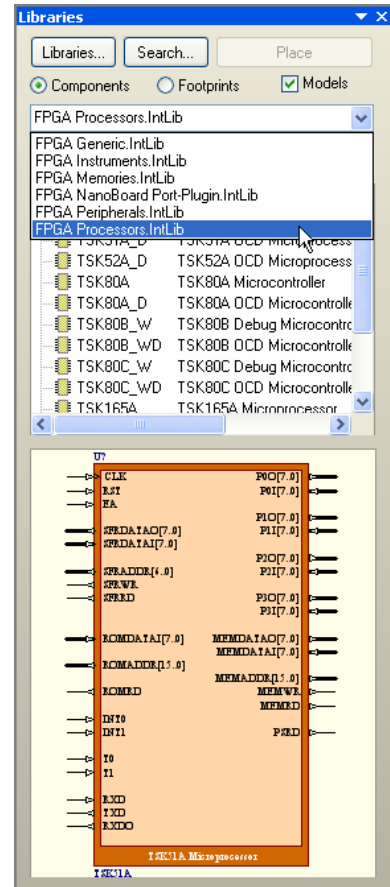


Figure 14. Place components from the FPGA-ready component libraries

For help on an FPGA-ready component, press F1 after clicking on the component in the list in the Libraries panel.

PS2 Controller – parallel to serial interface providing a bidirectional, synchronous serial interface between a host MCU and a PS/2 device (keyboard or mouse). The PS2 controller can be used in conjunction with either of the two sets of PS2 interface hardware on the NanoBoard.

SRL0 – simple parallel to serial interface, full duplex, single byte buffering. The SRL0 can be used in conjunction with the RS-232 interface hardware on the NanoBoard.

TMR3 – dual timer unit, 16, 13 and 8-bit timer/counter modes.

VGA – VGA controller that creates a simple method of implementing a VGA interface, presenting video memory as a flat address space. Supports VGA and SVGA resolutions, and B&W, 16 and 64 color. Outputs digital RGB and H+V sync. The VGA controller can be used in conjunction with the VGA output on the NanoBoard.



The on-line help system includes a hardware reference manual for each peripheral component, under FPGA design - Core References - Peripherals.

Generic components

Generic components can be placed from the library `\Program Files\Altium\Library\Fpga\FPGA Generic.IntLib`. This library is included to implement the interface logic in your design. It includes pin-wide and bus-wide versions for many components, simplifying the wiring complexity when working with buses. As well as a broad range of logic functions, the Generic library also includes pullup and pulldown components as well as a range of bus joiners, used to manage the merging, splitting and renaming of buses.



For a definition of the naming convention used in the generic library and a complete listing of available devices, refer to the [FPGA Generic Library Guide](#).



For information on working with buses and using bus joiners, refer to the topic, [Using buses and bus joiners](#) earlier in this document.

Vendor macro and primitive libraries

If vendor independence is not required, there are also complete Altera and Xilinx primitive and macro libraries. These libraries can be found in the respective Altera and Xilinx sub-folders in `\Program Files\Altium\Library\`. The macro and primitive libraries include the string 'FPGA' in the library name. Note that some vendors require you to use primitive and macro libraries that matches the target device. Designs that include vendor components cannot be re-targeted to another vendor.

Virtual Instruments

To test the state of internal nodes in the design you can 'wire in' virtual instruments. The 'hardware' portion of the instrument is placed and wired on the schematic like other components, and then synthesized into the FPGA. The interface to each instrument is accessed in the Devices view once the design has been synthesized and the FPGA programmed.



For information on working in the Devices view, refer to the application note [Processing the Captured FPGA Design](#).

The instrument hardware that has been synthesized into the FPGA communicates with its interface using the Nexus communications standard, over the JTAG link.



For information on the JTAG communications, refer to the application note [PC to NanoBoard Communications](#).



There is a reference manual for each instrument in the on-line help system, Select **Help » Contents** in the menus then navigate to FPGA Design - Core References - Instruments.

The following virtual instruments are available:

Digital I/O (IOB_1X8 thru IOB_4X16)



Figure 15. Digital IO module, used to monitor and control nodes in the design

The digital I/O is a general purpose tool that can be used for both monitoring and activating nodes in the circuit. It is available in either 8-bit wide or 16-bit wide variants, with 1 to 4 channels.

Each input bit presents as a LED, and the set of 8 or 16 bit also presents as a HEX value. Outputs can be set on a bit-basis by clicking the appropriate bit in the Outputs display, or a HEX value can be typed in the HEX field. If a HEX value is entered you must click the **>>** button to output it. The Synchronize button can be used to transfer the current input value to the outputs.

Frequency generator (CLKGEN)

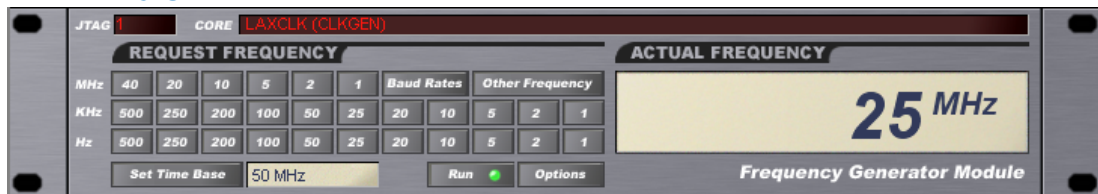


Figure 16. Frequency generator, used to generate the specified frequency

The frequency generator outputs a 50% duty cycle square wave, of the specified frequency. Predefined frequencies can be chosen by clicking the appropriate button, or any frequency can be defined using the **Other Frequency** button. If the specified frequency cannot be generated the closest possible is generated and the error shown on the display. Note that when the frequency generator is instantiated in the FPGA it will not be running, you must click the Run button to generate an output.

Frequency counter (FRQCNT2)



Figure 17. Frequency counter, used to measure frequency in the design

The frequency counter is dual input counter that can display the measured signal in 3 different modes, as a frequency, period, or number of pulses.

Logic Analyzer (LAX_1K8 thru LAX_16)



Figure 18. Logic analyzer instrument, with a logic analyzer component shown in the inset. Use the LAX to monitor multiple nets in the design, then display the results as a digital or an analog waveform.

The logic analyzer allows you to capture multiple snapshots of multiple nodes in your design. The available logic analyzers support the simultaneous capture of 8 or 16 nodes, or bits. The number of capture snapshots is defined by the amount of capture memory, this ranges from 1K to 4K of internal storage memory (using internal FPGA memory resources). There is also a 8-bit and a 16-bit external memory variants.



For more detailed information on using the logic analyzer, refer to the [Logic Analyzer](#) reference manual.

Waveform display features

The capture results are displayed in the instrument panel. There are also two waveform display modes. The first is a digital mode, where each capture bit is displayed as a separate waveform and the capture events define the timeline. Note that the capture clock must be set in the logic analyzer options for the timeline to be calculated correctly. Click the **Show Digital Waves** button to display the digital waveform.

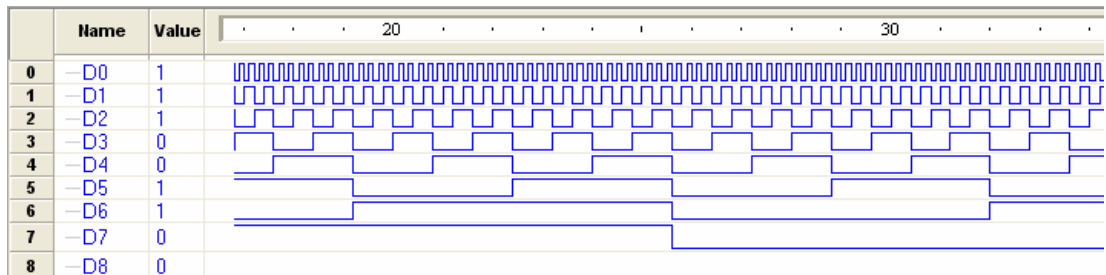


Figure 19. Digital waveform capture results from the logic analyzer

The second waveform mode is an analog mode, where the value on all the logic analyzer inputs is displayed as a voltage, for each capture event. The voltage range is from zero to the maximum possible count value, scaled to a default of 3.3V (defined in the *Logic Analyzer Options* dialog). Click the **Show Analog Waves** button to display the analog waveform.

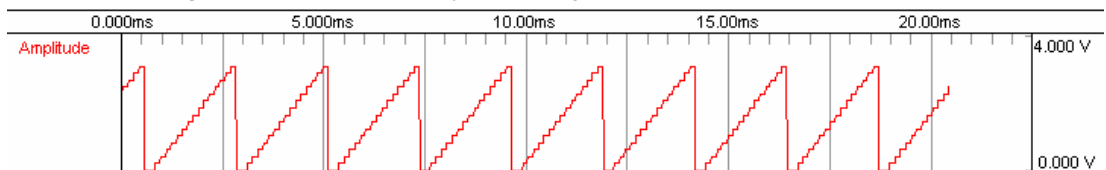


Figure 20. Analog waveform capture results from the logic analyzer

Continuous Display Mode

Note that updates performed in the logic analyzer panel are displayed immediately as a waveform, allowing you to interactively examine capture results. There is also a continuous display mode, enable/disable this using the buttons on the toolbar in either the digital or analog wave display window.

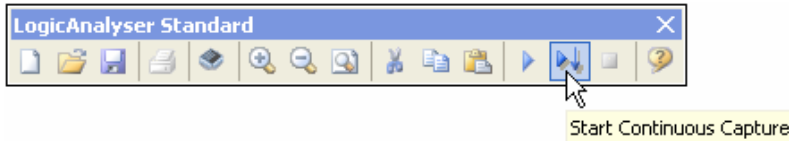


Figure 21. Enabling the continuous capture mode.

Implementing the JTAG for the processors and instruments

Communications from the DXP software environment to the embedded processors and virtual instruments is done over a JTAG communications link, referred to as the *soft devices chain*, displayed as the 3rd chain in the Devices view in the software. If your design includes a processor or an instrument you must enable the soft devices chain by placing the following components on the top sheet of your design.

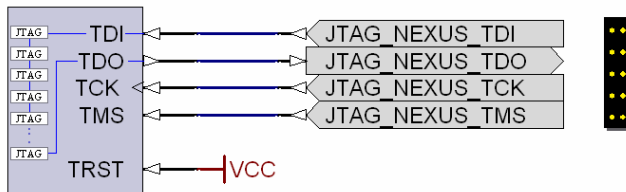


Figure 22. the soft devices JTAG chain is implemented by placing the NEXUS_JTAG_PORT (on the left) from the FPGA Generic library and the NEXUS_JTAG_CONNECTOR from the FPGA NanoBoard Port-Plugin library.

For an overview of the Devices view refer to the [Devices view](#) section later in this guide.

For information on the JTAG chains, refer to the application note [PC to NanoBoard Communications](#).

External components that are on the NanoBoard

The NanoBoard includes a variety of useful input and output components connected to I/O pins on the FPGA. Normally you use Ports to connect from the nets in a design to the pins on the FPGA. However, since the connectivity from the FPGA to the components on the NanoBoard is fixed by the routing there is no need to place ports and then define the net-to-pin mapping. Instead there is a library of special components that can be placed instead, these components are in the FPGA NanoBoard Port Plug-in library.

These components are placed on the top sheet, instead of ports. They are recognized as being external to the FPGA design by the presence of the **PortComponent = True** parameter in each component, and are automatically converted to ports during synthesis.

Refer to the [NanoBoard Technical Reference Manual](#) for more information on using the features on the NanoBoard.

Embedded software development

The Nexar product includes complete software development tool chains for all supplied processor cores. Using Altium's TASKING Viper compiler technology, Nexar provides high-quality code development and debugging that is fully integrated in the DXP environment.

Once the target design has been downloaded to the NanoBoard, all processors in the system can be controlled and debugged from within the environment. This enables software development to take place directly on the target hardware from early in the design cycle, supporting parallel development of hardware and software.

The Viper compiler technology also supports multi-core debugging, allowing simultaneous debugging of multiple processors inside an FPGA.

The Embedded project

Like all DXP-based projects, the embedded project file is an ASCII file that stores links to source code files, compiler settings, and so on. Create the embedded project file (PrjEmb), save it, then add source code files by clicking the **Project** button in the *Projects* panel. Embedded project options, including compiler, assembler, linker, optimization and build options, are defined in the **Options for Project** dialog.

The coding environment

Code editing is performed within the DXP environment. Syntax aware code editors support multiple languages, including TSK165, TSK51 and TSK80 assembler, as well as C (C compilers are included for the TSK51 and TSK80). The coding environment supports all the advanced features expected in a professional embedded software development environment, including:

- Project management
- Extended syntax highlighting, including function recognition
- Code collapse, with reveal on hover feature
- Built in code formatter, reformats existing code using user-definable specifications
- Integrated debugging, run directly from the source code editor
- Code explorer, allowing easy navigation of the embedded project
- Tip on hover, displays declarations when not debugging and current value during debug

Compiling the project

Project compilation is performed from within the coding editor, select **Compile** from the **Project** menu. Compiler options, such as memory model, optimization settings and language options are configured in the **Options for Project** dialog. Warnings and errors generated during compile are displayed in the Messages panel, double click a message to cross probe to the source code.




For more information on the compiler, refer to the topic **Using the Compiler** in the **Embedded Software Development** section of the on-line help.

Simulating and debugging

Simulation and debugging is performed directly from within the code editor, launch a simulation or debug session from the **Debug** menu, or right click on the Project name in the **Projects** panel.

The Viper compiler/debugger technology supports multi-core debugging, allowing simultaneous debugging of multiple processors in an FPGA design. The debugger also supports:

- Breakpoints, in both the source view and the disassembly view
- Conditionals on breakpoints
- Pass count breakpoints
- Disassembly view with source and address breakpoints in both the mixed and pure disassembly modes
- Registers panel
- Watches panel
- Locals panel
- Call stack panel
- Memory space panels
- Debug console

 For more information on the debugger, refer to the topic **Using the Debugger** in the **Embedded Software Development** section of the on-line help.

Real-Time Operating System

The TDK51 includes a compact RTOS, compliant with the OSEK/VDX standard. The RTOS is a real-time, preemptive, multitasking kernel, designed for time-critical embedded applications. It offers:

- A high degree of modularity and the ability to create flexible configurations
- Time critical support, through the use of system object creation during the system generation phase
- Well defined interfaces between application software and the operating system
- Superior application software portability, via the use of the OSEK Implementation Language, or OIL

The RTOS panel is a runtime status panel, which can display information such as System Status, Alarms, Tasks and Resources. Open the RTOS panel via the **Embedded** button at the bottom right of the workspace, then enable the required RTOS information by clicking the **RTOS** button on the Debug toolbar.

 For more information on the RTOS, refer to the [8051 RTOS](#) guide.

Accessing the embedded code debugging panels

The embedded tools make extensive use of panels, click the **Embedded** button at the bottom right of the workspace to display a panel.

Linking the embedded project to the hardware design

You link the embedded project to the processor that it runs on by making it a sub-project of the FPGA hardware design. This is done in the Structure Editor, a display mode of the **Projects** panel. Click the Structure Editor option near the top of the panel to enable it.

The Structure Editor is used for linking sub-projects to parent projects in the following situations:

- Embedded project to the processor on the FPGA project
- Core project to a core component
- FPGA project to the PCB project that the device is used on

Using the Structure Editor to link

The upper region of the Structure Editor displays open projects, and their current relationship. If a sub-project is already linked it will be shown within the tree of the parent project, if it is not linked then it is shown as a separate project.

Linkage is created and broken using drag and drop. When you click and hold on a sub-project all possible drop locations (valid linkage points) will highlight in blue, simply drop the sub-project onto the parent project to create the linkage.

To break the linkage, drag the sub-project away from the parent project and drop it on a clear region of the Structure Editor.

The linkage can also be examine in the *Component Properties* dialog of the processor component that the embedded software project is linked to.

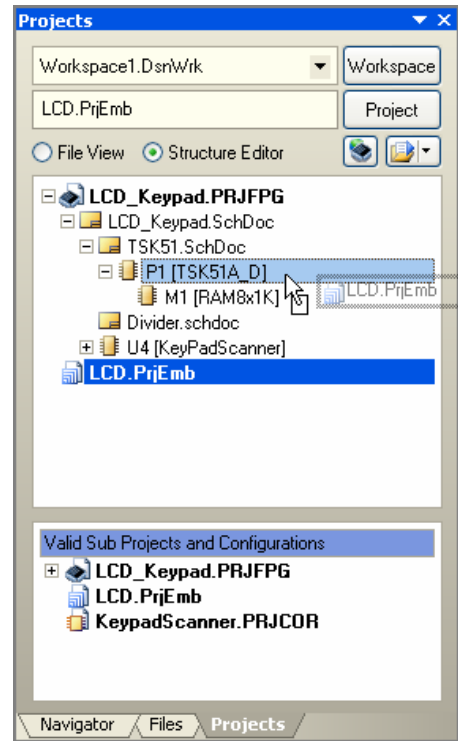


Figure 23. Use the Structure Editor to link the embedded project to the processor in the FPGA hardware design.

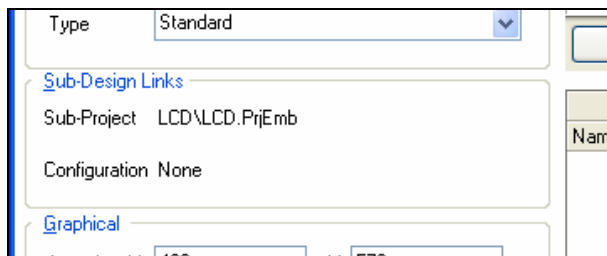


Figure 24. The name and location of the linked sub-project is displayed in the Component Properties dialog, of the processor that the embedded code runs on.

Configuring the design for the target FPGA

Once the design is captured you are ready to synthesize, perform a place and route in the vendor tools, and download the design to the NanoBoard. Before a synthesis can be performed you must include information that maps the design to the target device on the board.


The process of mapping or constraining the design to its physical implementation is done by creating constraint files – files that specify implementation detail such as the target device, the net-to-pin mapping, pin voltage requirements, and so on. The minimum information required to synthesize the design is the device specification.

When the design is going on the NanoBoard

Setting up to implement the design on the NanoBoard is quite straightforward. The system includes a constraint file for each supported device. Add these to your project (right click on the project filename in the **Projects** panel and select **Add Existing to Project**), NanoBoard constraint files are in:

```
C:\Program Files\Altium2004\Library\Fpga
```


Once the constraint files have been added, you need to create a configuration (a configuration is simply a defined set of constraint files). To add a new configuration right click on the project filename again, and select **Configuration Manager**. Add a new configuration, assign the constraint file for the target device, and you are ready to process the design and download onto the NanoBoard.


 There is an example tutorial that goes through this process in detail, refer to [Getting Started with FPGA Design](#) for more information.

When the design is targeting your own board

To target the design to your own board you need to:


1. Create a constraint file. Right click on the project filename in the **Projects** panel and select **Add New to Project » Constraint File**. In the constraint file editor you can select the target device, amongst other things.
2. Set up a configuration. Right click on the project filename and select **Constraint Manager**. Add a new configuration, and assign your constraint file.

 For details about creating your own constraint file and getting to synthesis, refer to the application note [Re-targeting the design to the Production Board](#).

 For a detailed description of configurations and constraints, and their role in design portability, refer to the article [Design Portability, Configurations and Constraints](#).

Specifying design constraints

There are numerous constraints that you might need to include in your design, such as pin mapping, pin IO standard, drive current, clock requirements, and so on. Constraints can be included in the constraint file, or on the design as parameters.

 For details about supported constraints, refer to the [Constraint File Reference](#).

Processing and implementing the captured design

So far this document has given an overview of the 2 main sections of the design process – capturing the 'hardware' design, and writing the embedded software that runs on the processor in that hardware design. Figure 25 shows the logical flow of these 2 processes.

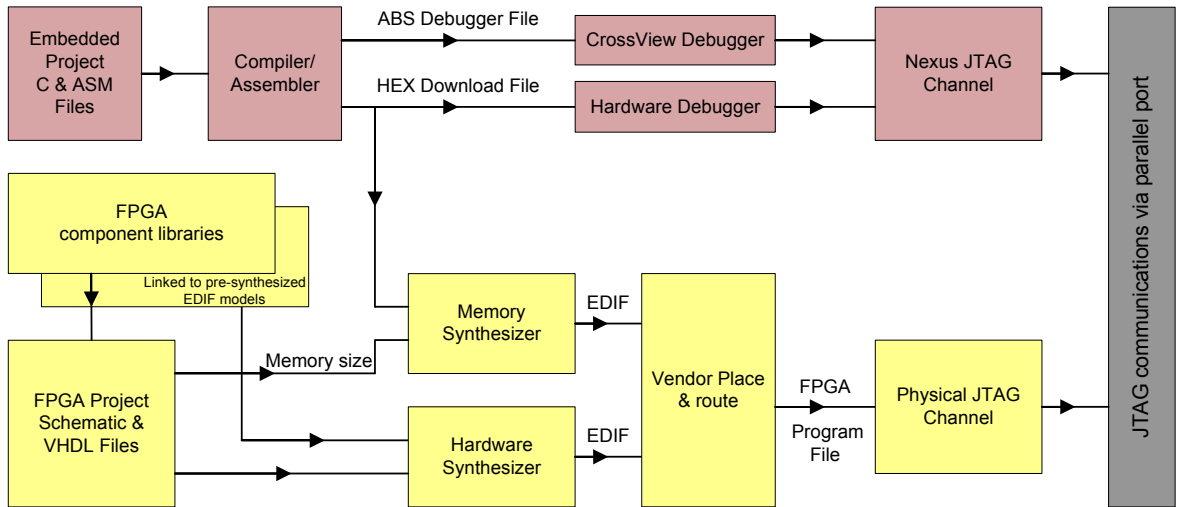


Figure 25. The flow of the embedded software development and the hardware design

Before the design can be implemented in an FPGA there are a number of steps that must be carried out. These include:

Compiling – this process analyzes the design structure and checks for numerous design errors, such as connectivity.

Synthesizing – this process translates the high-level source design files into a low level description, capable of being read by vendor place and route tools.

Building – this is the process of implementing the design in the target FPGA. It requires that you have appropriate vendor place and route tools installed on your PC, such as Altera's Quartus, or Xilinx's ISE (both of these tools are available in free webpack versions, download from www.altera.com or www.xilinx.com).

These steps are all performed in the Devices view. If you have a NanoBoard connected to the parallel port on your PC when you open the Devices view, the NanoBoard and the FPGA mounted on it will appear as shown in the upper part of Figure 26. Once the FPGA design has been processed any processors or virtual instruments in the design will appear in the soft chain.

Processing a design without a NanoBoard

If you do not have a NanoBoard connected to your PC you can manually add an FPGA into the Devices view. This will allow you to perform a build and confirm that the design can be implemented in the chosen device. Right-click in the Devices view and select **Add » Browse** to select a device in the *Choose Physical Device* dialog.

The Devices view – managing the process flow

The Devices view (**View » Devices**) provides the central location to control the process of taking the design from the capture state through to implementing it in an FPGA.

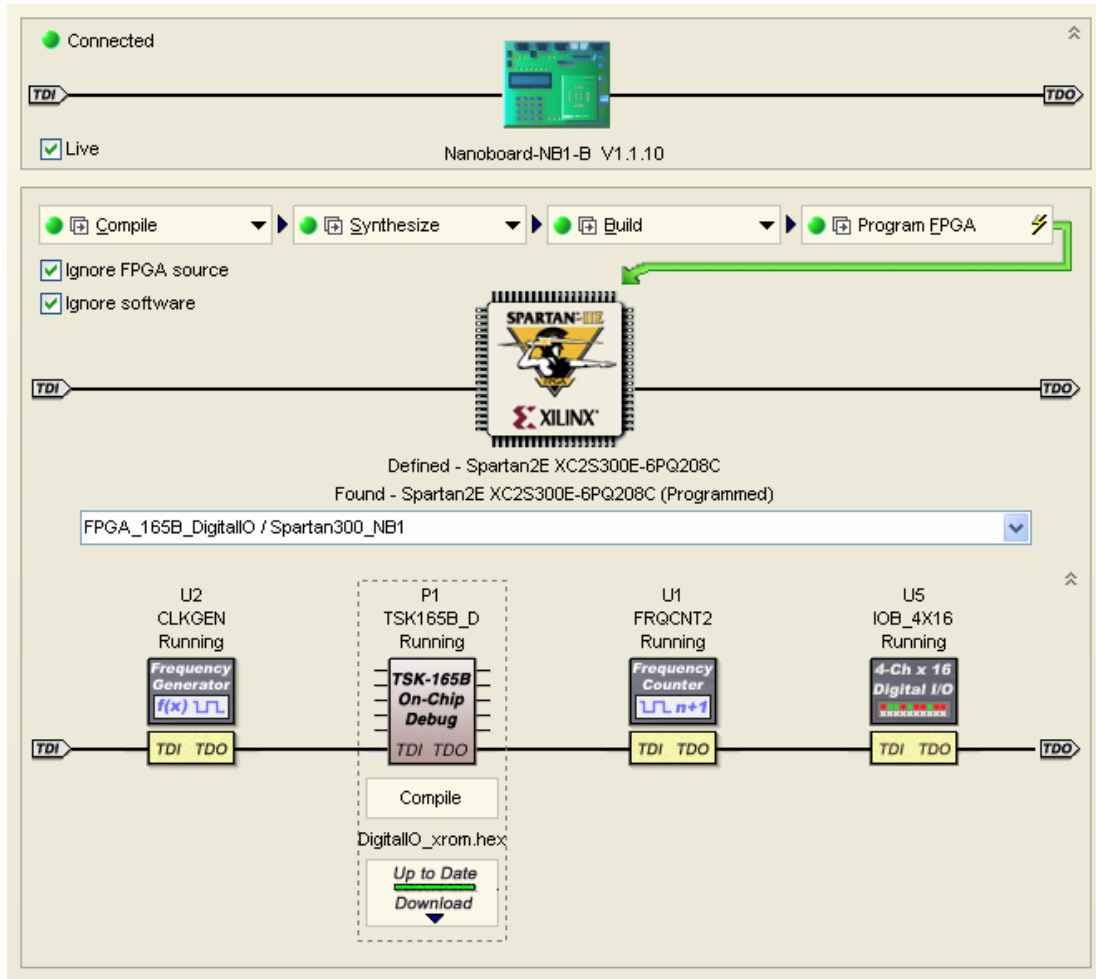


Figure 26. The Devices view, used to process and implement the design on the NanoBoard, then communicate with it during debugging.

What you see in the Devices view

The devices view shows 3 horizontal black lines, each of these represents a JTAG communications chain. Software-to-NanoBoard communications are all performed using JTAG as the communications mechanism, with the physical PC-to-board communications being carried out via the parallel port on the PC. Each of the 3 JTAG chains offers a different set of features.

JTAG, or boundary scan, is a standard initially developed for testing ICs mounted on a PCB. It has since become the 'carrier' for high-level product development communications systems, such as the Nexus embedded debugging standard.

The NanoBoard chain – The first chain is referred to as the NanoBoard chain. It provides access to the NanoBoard features including the programmable clock and the SPI configuration devices. Double click on the board icon to display the *NanoBoard Controller* instrument and configure one of these features.

The hard devices chain – The second chain is referred to as the hard devices chain. It shows all JTAG compliant devices on the board that are currently accessible in the chain. The buttons above the chain are used to process the design and program the FPGA, below the chain is the name of the project/configuration combination that is currently targeted to the FPGA. The status of this project/configuration is reflected by the color of the indicators in the Compile, Synthesize, Build and Program FPGA buttons.

This chain is also used to enable the Live Update feature that performs boundary scan monitoring of the status of component pins. Double click on the FPGA icon to display the *Hard Devices* instrument and enable **Live Update**.

The soft devices chain – The third chain is referred to as the soft devices chain. It shows all Nexus compatible devices that have been implemented inside the FPGA, including processors and virtual instruments. Double click on an MCU or virtual instrument to open an instrument panel to control that MCU or instrument.

What you can do in the Devices view

From the Devices view you can compile and synthesize the hardware design, perform an FPGA place and route, and download the design into the target FPGA. You also have access to the embedded software development tools.



For a detailed description of working in the Devices view, refer to the application note [Processing the Captured FPGA Design](#).

Configuring the NanoBoard controller

Double-click on the NanoBoard icon at the top of the Devices view to open the NanoBoard controller instrument. Here you can set the frequency of the programmable clock and program the 2 serial flash RAM devices on the NanoBoard. One of the flash RAM components can be used to store the configuration image of your design and program the NanoBoard FPGA on power up (fit the `Auto Load FPGA` link on the NanoBoard), the other can be accessed from within your design as general purpose serial storage (place the `SerialFMemory` component from the NanoBoard Port Plugin library).



Figure 27. Use the NanoBoard controller to set the clock frequency and program the 2 flash RAM devices

Compile – pre-synthesis verification

Since synthesis places strict requirements on design interfaces – such as I/O types declared in ports and sheet entries – it is important that the design is compiled and all errors and warnings are resolved prior to synthesizing the design.

FPGA designer's QuickStart guide

Clicking the **Compile** button performs a structural compile of the project, establishing sheet-to-sheet connectivity and displaying the project structure in the **Navigator** panel.

Once the design has been compiled, use the **Messages** panel to examine any errors or warnings – double click to cross probe to the cause of the problem. Error checks are configured in the *Options for Project* dialog (**Projects** menu).

Design synthesis

Synthesis is the process of converting the high level schematic / behavioral VHDL description to a low-level gate description, suitable for processing by the FPGA vendor place and route tools. The built-in DXP synthesis engine first produces an hierarchical VHDL netlist of the design, which is then synthesized into an EDIF description. As well as the EDIF file that describe the design connectivity, the pre-synthesized EDIF description of each component is also copied into the project sub-folder that is automatically created under the project folder.

Synthesis options are configured in the *Options for Project* dialog (**Projects** menu).

Build – vendor tools place and route

The build stage first generates the file set required by the vendor place and route software, including project and constraint files. It then runs the vendor tools to process the EDIF description and perform a place and route, timing analysis, and generate the device program file.

While the default settings will be suitable for most designs, you have access to many of the vendor tool configuration options by clicking the Options icon next to each stage of the build process.

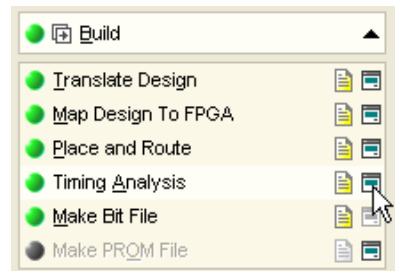


Figure 28. Click the Options icons to configure that stage of the Build



You can also include vendor constraint files in your design, refer to the specific working with vendor tools documents, including [Working with Altera Devices and Place and Route Tools](#) and [Working with Xilinx Devices and Place and Route Tools](#) for more information.

Program FPGA

This button downloads the device program file to the device. The download progress is displayed on the status bar.

Monitoring the state of the FPGA pins

Once the design has been downloaded to the FPGA, the hard devices chain can be used to monitor the state of the FPGA pins. To do this double-click on the FPGA icon in the Devices view to open the *Hard Devices* instrument, then enable the **Live Update** option.

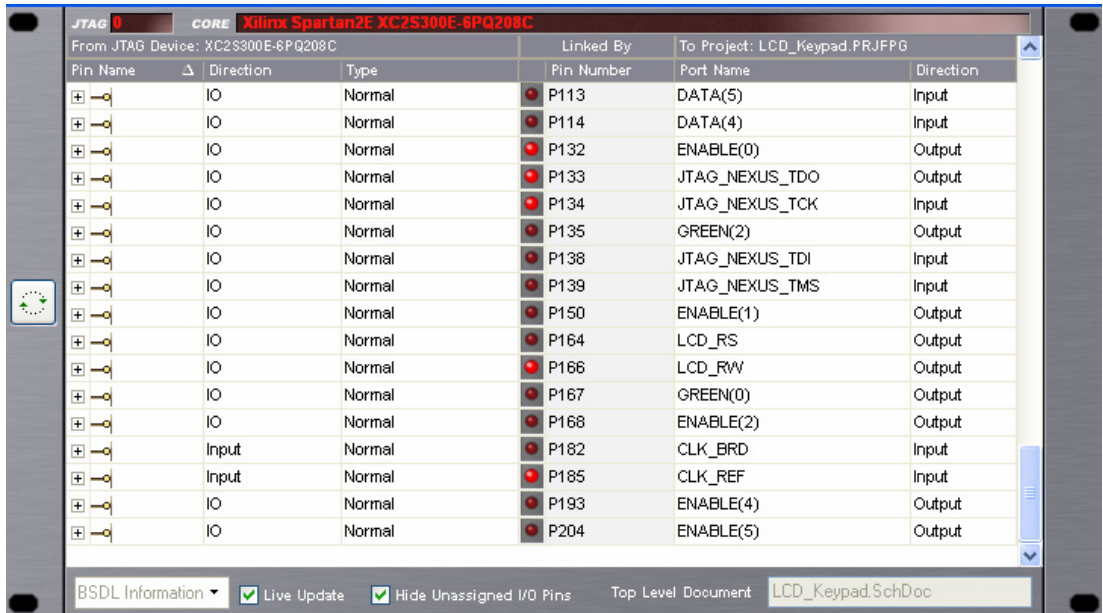


Figure 30. Enable the Live Update option in the hard Devices instrument to monitor the state of the FPGA pins.

The LED indicators in the panel will display the current state of the FPGA pins. You can also monitor the pin status back on the schematic, place a Probe object on any net that connects to an FPGA pin to see the current status of that net or bus (**Place » Directive » Probe**).

Note that the *Hard Devices* instrument must remain open for this feature to function, and the source design must be compiled.

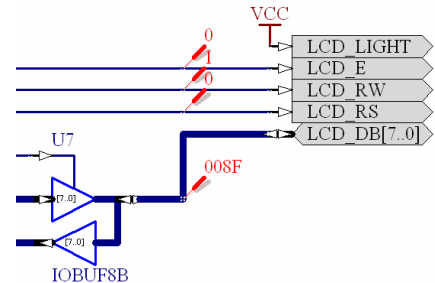


Figure 29. Place probes on the schematic.

Working with an embedded processor

If your design includes a processor, the devices view gives access to embedded software tool features. Right-click on the processor icon to pause or reset the processor, or to launch a debug session. Refer to the [Embedded software development](#) section of this document for more information.

Working with multiple FPGAs

The FPGA design environment supports the simultaneous development of FPGAs on multiple NanoBoards or connected user boards. If you have multiple FPGAs present in the devices view, you must have a valid design downloaded into each device to use the soft devices chain. If one FPGA in this chain includes soft devices and others do not, each design that does not include soft devices must include the 2 JTAG implementation components, as described in the section [Implementing the JTAG for the processors and instruments](#), elsewhere in this document.

Testing and debugging the design

Traditionally FPGAs are designed using a hardware description language and verified using an HDL simulator against an HDL testbench. From there the design is implemented in an FPGA, either on a development board or a prototype board, and traditional hardware design debugging techniques are used to verify that the device performs as predicted by the simulation.

Without a design environment that support in-circuit testing of the design inside an FPGA it is difficult to debug the FPGA design any other way. Add to this the lack of design environments that give embedded software development tools access to a processor running inside an FPGA and the result is that to date, FPGAs have been limited to implementing specialized components in a larger digital system, and their testing relies heavily on simulation and verification.

As well as a lack of tools that support debugging the embedded processor, the traditional verification model of HDL simulation does not lend itself to testing and debugging a design that includes a processor.

A proven approach to test and debug

A digital system that includes a processor, peripherals and discrete logic mounted as components on a PCB is typically tested and debugged by:

- Running test software on the processor, via an in-circuit emulator or equivalent software debugger
- Attaching a logic analyzer to monitor the state of buses in the design
- Using an oscilloscope to monitor specific nodes in the circuit and verify signals against the original design

Rather than attempting to write the source HDL so that it is both simulation and synthesis ready, and then simulate the entire digital system, the DXP based FPGA design environment supports the same test and debug methodology used in traditional PCB development.

It includes:

- Embedded software debug tools
- Instruments, including logic analysers
- Boundary scan monitoring that show the state of pins on the FPGA
- The NanoBoard – an implementation platform that allows the design to be tested before moving it to the final PCB, and then supports integrated testing of the design on the final PCB.

Rapid design iterations

Using this approach, the design can be implemented in the target device and tested extensively, before moving it to the final PCB. Design iterations can be performed in minutes, and design options, such as hardware / software partitioning can be explored, all before committing the design to a PCB.

Linking to the PCB design project

Another advantage of implementing a design in an FPGA is that the device pins that the internal signals connect to are, to a large extent, user definable. This can result in significant improvements to the route-ability of the PCB, potentially reducing the cost and complexity.

This of course must be traded against successfully placing and routing the design in the FPGA, so there may be a number of iterations taken to ensure that design requirements on both the PCB and FPGA projects are met.

Creating the FPGA project to PCB project link

To link an FPGA project to the device on the PCB that is implemented in, use the Structure Editor in the **Projects** panel to create the sub-project to project linkage.

Maintaining the FPGA project to PCB project link

The system is capable of compiling both the PCB and FPGA designs and establishing, via net names, the connectivity between them. An overview of the current state of the connectivity is presented when you open the *FPGA Workspace Map* (**Project** menu).

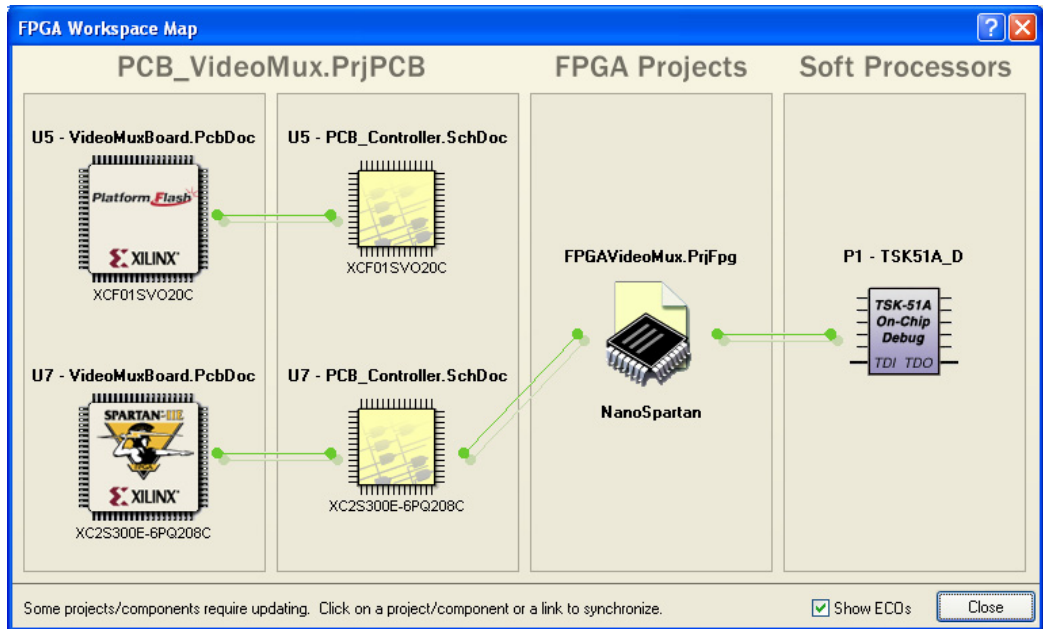


Figure 31. Use the Workspace map to manage the FPGA to PCB project linkage.



For complete details on creating and maintaining the FPGA to PCB project linkage and performing PCB pin optimizations, refer to the application note, [Linking an FPGA Project to a PCB Project](#).

Revision History

Date	Version No.	Revision
26-Jan-04	1.0	New product release

Software, documentation and related materials:

Copyright © 2004 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, CAMtastic, CircuitStudio, Design Explorer, DXP, LiveDesign, NanoBoard, NanoTalk, Nexar, nVisage, P-CAD, Protel, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.