



Reed-Solomon Compiler

User Guide



101 Innovation Drive
San Jose, CA 95134
www.altera.com

MegaCore Version: 9.0
Document Date: March 2009

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Chapter 1. About This Compiler

Release Information	1-1
Device Family Support	1-1
Features	1-2
General Description	1-2
OpenCore Plus Evaluation	1-3
DSP Builder Support	1-3
Performance and Resource Utilization	1-4

Chapter 2. Getting Started

Design Flow	2-1
RS Compiler Walkthrough	2-2
Create a New Quartus II Project	2-2
Launch IP Toolbench	2-3
Step 1: Parameterize	2-4
Step 2: Set Up Simulation	2-7
Step 3: Generate	2-8
Simulate the Design	2-10
Compile the Design	2-11
Program a Device	2-11
Set Up Licensing	2-12

Chapter 3. Functional Description

Background	3-1
Erasures	3-2
Shortened Codewords	3-2
Variable Encoding & Decoding	3-3
RS Encoder	3-3
RS Decoder	3-4
Error Symbol Output	3-5
Bit Error Count	3-6
Interfaces	3-6
OpenCore Plus Time-Out Behavior	3-7
Parameters	3-8
Signals	3-9
Throughput Calculator	3-11

Appendix A. Using the RS Encoder or Decoder in a CCSDS System

Introduction	A-1
Test Patterns	A-1

Additional Information


Revision History	Info-1
How to Contact Altera	Info-1
Typographic Conventions	Info-1

Release Information

Table 1–1 provides information about this release of the Reed-Solomon (RS) Compiler.

Table 1–1. RS Compiler Release Information

Item	Description
Version	9.0
Release Date	March 2009
Ordering Codes	IP-RSENC (Encoder) IP-RSDEC (Decoder)
Product IDs	0039 0041 (Encoder) 0080 0041 (Decoder)
Vendor ID	6AF7

 For more information about this release, refer to the [MegaCore IP Library Release Notes and Errata](#).

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore® function. The [MegaCore IP Library Release Notes and Errata](#) report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release."

Device Family Support

MegaCore functions provide either full or preliminary support for target Altera® device families, as described below:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and may be used in production designs
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it may be used in production designs with caution

Table 1–2 shows the level of support offered by the RS Compiler to each of the Altera device families.

Table 1–2. Device Family Support (Part 1 of 2)

Device Family	Support
Arria™ GX	Full
Arria II GX	Preliminary
Cyclone®	Full
Cyclone II	Full
Cyclone III	Full

Table 1–2. Device Family Support (Part 2 of 2)

Device Family	Support
HardCopy® II	Full
HardCopy III	Preliminary
HardCopy IV E	Preliminary
Stratix®	Full
Stratix II	Full
Stratix II GX	Full
Stratix III	Full
Stratix IV	Preliminary
Stratix GX	Full

Features

- High-performance encoder/decoder for error detection and correction
- Fully parameterized RS function, including:
 - Number of bits per symbol
 - Number of symbols per codeword
 - Number of check symbols per codeword
 - Field polynomial
 - First root of generator polynomial
 - Space between roots in generator polynomial
- Decoder features:
 - Variable option
 - Erasures-supporting option
- Encoder features variable architectures
- Support for shortened codewords
- Conforms to Consultative Committee for Space Data Systems (CCSDS) Recommendations for Telemetry Channel Coding, May 1999
- Easy-to-use IP Toolbench interface:
 - Generates parameterized encoder or decoder
 - Generates customized testbench and customized Tcl script
- DSP Builder ready
- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Support for OpenCore Plus evaluation

General Description

The Altera RS Compiler comprises a fully parameterizable encoder and decoder for forward error correction applications. RS codes are widely used for error detection and correction in a wide range of DSP applications for storage, retrieval, and transmission of data. The RS Compiler has the following options:

- Erasures-supporting option—the RS decoder can correct symbol errors up to the number of check symbols, if you give the location of the errors to the decoder (see [“Erasures” on page 3–2](#)).
- Variable encoding or decoding—you can vary the total number of symbols per codeword and the number of check symbols, in real time, from their minimum allowable values up to their selected values, when you are encoding or decoding.
- Error symbol output—the RS decoder finds the error values and location and adds these values in the Galois field to the input value.
- Bit error output—either split count or full count

OpenCore Plus Evaluation

With Altera’s free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPPSM megafunction) within your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include megafunctions
- Program a device and verify your design in hardware

You only need to purchase a license for the megafunction when you are completely satisfied with its functionality and performance, and want to take your design to production.



For more information on OpenCore Plus hardware evaluation using the RS Compiler, see [“OpenCore Plus Time-Out Behavior” on page 3–7](#) and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

DSP Builder Support

Altera’s DSP Builder shortens DSP design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

You can combine existing MATLAB/Simulink blocks with Altera DSP Builder/MegaCore blocks to verify system level specifications and perform simulation. After installing this MegaCore function, a Simulink symbol of this MegaCore function appears in the Simulink library browser in the MegaCore library from the Altera DSP Builder blockset.



When using the RS MegaCore function in Simulink with DSP Builder, the IO ports of the RS DSP Builder block are represented as unsigned integer data type. Therefore, when you want to connect a signal with a non-unsigned integer data type to the RS DSP Builder block IO port, a DSP Builder casting block such as the “Bus Conversion Block” must be inserted to convert the signal to unsigned integer.



For more information on DSP Builder, refer to the *DSP Builder User Guide* and the *DSP Builder Reference Manual*.

Performance and Resource Utilization

Table 1-3 shows the typical performance using the Quartus II software for Cyclone III (EP3C10F256C6) devices.

Table 1-3. Performance—Cyclone III Devices

Parameters					Combinational LUTs	Logic Registers	Memory (M9K)	f _{MAX} (MHz)	Throughput (Mbps)
Options	Keysize	m	N	check					
None	half	4	15	6	541	365	5	230	216
None	half	8	204	16	1,720	995	5	202	1,613
Split bit error output	half	8	204	16	1,765	1,057	5	194	1,552
Full bit error output	half	8	204	16	1,778	1,058	5	190	1,519
None	half	8	255	32	2,972	1,676	5	193	1,213
Variable	half	8	204	16	1,886	1,074	5	202	1,620
Erasures	half	8	204	16	3,151	1,561	5	188	1,500
Erasures and variable	half	8	204	16	3,465	1,704	6	191	1,527
None (1)	—	8	204	16	256	210	—	324	2,593
Variable (1)	—	8	204	16	1,048	313	—	237	1,897
Variable (1)	—	8	204	32	2,341	580	—	227	1,813

Note to Table 1-3:

(1) Encoder.

Table 1-4 shows the typical performance using the Quartus II software for Stratix III (EP3SE50F780C2) devices.

Table 1-4. Performance—Stratix III Devices (Part 1 of 2)

Parameters					Combinational ALUTs	Logic Registers	Memory (M9K)	f _{MAX} (MHz)	Throughput (Mbps)
Options	Keysize	m	N	check					
None	half	4	15	6	417	366	5	403	378
None	half	8	204	16	1,139	998	5	358	2,865
Split bit error output	half	8	204	16	1,196	1,060	5	336	2,686
Full bit error output	half	8	204	16	1,181	1,065	5	328	2,624

Table 1-4. Performance—Stratix III Devices (Part 2 of 2)

Parameters					Combinational ALUTs	Logic Registers	Memory (M9K)	f _{MAX} (MHz)	Throughput (Mbps)
Options	Keysize	m	N	check					
None	half	8	255	32	2,027	1,685	5	319	2,011
Variable	half	8	204	16	1,273	1,082	5	359	2,871
Erasures	half	8	204	16	2,092	1,564	5	309	2,469
Erasures and variable	half	8	204	16	2,200	1,708	6	311	2,490
None (1)	—	8	204	16	204	210	—	621	4,969
Variable (1)	—	8	204	16	779	313	—	397	3,179
Variable (1)	—	8	204	32	1,650	581	—	365	2,923

Note to Table 1-4:

(1) Encoder.

Table 1-5 shows the typical performance using the Quartus II software for Stratix IV (EP4SGX70DF29C2X) devices.

Table 1-5. Performance—Stratix IV Devices

Parameters					Combinational ALUTs	Logic Registers	Memory		f _{MAX} (MHz)	Throughput (Mbps)
Options	Keysize	m	N	check			ALUTs	M9K		
None	half	4	15	6	426	382	8	3	413	387
None	half	8	204	16	1,220	1,034	64	3	368	2,945
Split bit error output	half	8	204	16	1,273	1,092	64	3	340	2,719
Full bit error output	half	8	204	16	1,255	1,092	64	3	325	2,603
None	half	8	255	32	2,100	1,713	64	3	324	2,038
Variable	half	8	204	16	1,362	1,119	64	3	356	2,850
Erasures	half	8	204	16	2,170	1,596	64	3	314	2,510
Erasures and variable	half	8	204	16	2,322	1,746	96	3	310	2,480
None (1)	—	8	204	16	204	210	—	—	620	4,960
Variable (1)	—	8	204	16	777	313	—	—	387	3,099
Variable (1)	—	8	204	32	1,651	582	—	—	347	2,775

Note to Table 1-4:

(1) Encoder.

The throughput in megabits per second (Mbps) is derived from the formulas in Table 3-9 on page 3-11 and maximum frequency at which the design can operate.

Overall resource requirements vary widely depending on the parameter values used. The number of logic elements (LEs) or combinational ALUTs required to implement the function is linearly dependent on both the field size and the number of check symbols. More memory is required for 9, 10, 11, or 12 bits per symbol. Specifying the erasures-supporting and the variable option also increases the memory required.

Design Flow

To evaluate the Reed-Solomon Compiler using the OpenCore Plus feature, include these steps in your design flow:

1. Obtain and install the Reed-Solomon Compiler.

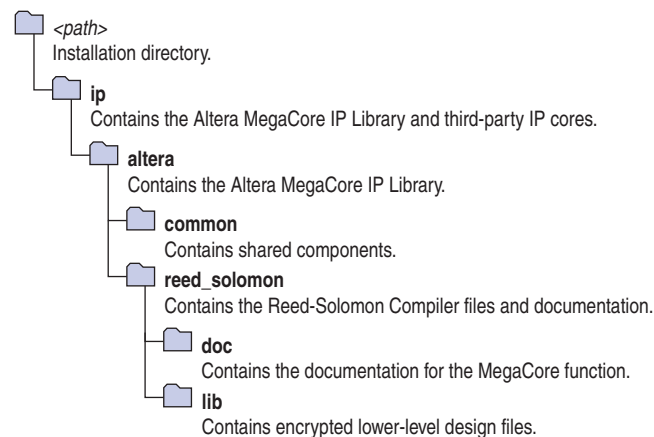
The Reed-Solomon Compiler is part of the MegaCore® IP Library, which is distributed with the Quartus® II software and downloadable from the Altera® website, www.altera.com.



For system requirements and installation instructions, refer to *Quartus II Installation & Licensing for Windows and Linux Workstations*.

Figure 2–1 shows the directory structure after you install the Reed-Solomon Compiler, where *<path>* is the installation directory. The default installation directory on Windows is `c:\altera\90`; on Linux it is `/opt/altera90`.

Figure 2–1. Directory Structure



2. Create a custom variation of the RS Compiler using IP Toolbench.



IP Toolbench is a toolbar from which you quickly and easily view documentation, specify parameters, and generate all of the files necessary for integrating the parameterized MegaCore function into your design.

3. Implement the rest of your design using the design entry method of your choice.
4. Use the IP Toolbench-generated IP functional simulation model to verify the operation of your design.



For more information on IP functional simulation models, see the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

5. Use the Quartus II software to compile your design.



You can also generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

6. Purchase a license for the RS Compiler.

After you have purchased a license for the RS Compiler, follow these additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera® device(s) on your board.
3. Program the Altera device(s) with the completed design.

RS Compiler Walkthrough

This walkthrough explains how to create an RS MegaCore function using the Altera RS Compiler IP Toolbench and the Quartus II software on a PC. As you go through the wizard, each step is described in detail. When you are finished generating a custom variation of the RS MegaCore function, you can incorporate it into your overall project.



IP Toolbench only allows you to select legal combinations of parameters, and warns you of any invalid configurations.

This walkthrough requires the following steps:

- “Create a New Quartus II Project” on page 2-2
- “Launch IP Toolbench” on page 2-3
- “Step 1: Parameterize” on page 2-4
- “Step 2: Set Up Simulation” on page 2-7
- “Step 3: Generate” on page 2-8


Create a New Quartus II Project

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity.


To create a new project follow these steps:

1. Choose **Programs > Altera > Quartus II <version>** (Windows Start menu) to run the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.
2. Choose **New Project Wizard** (File menu).
3. Click **Next** in the **New Project Wizard Introduction Page** (the introduction page does not display if you turned it off previously).

4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:
 - a. Specify the working directory for your project. For example, this walkthrough uses the **c:\altera\projects\rs_project** directory.

 The Quartus II software automatically specifies a top-level design entity that has the same name as the project. This walkthrough assumes that the names are the same

- b. Specify the name of the project. This walkthrough uses **project** for the project name.
5. Click **Next** to close this page and display the **New Project Wizard: Add Files** page.

 When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

6. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.
7. On the **New Project Wizard: Family & Device Settings** page, choose the target device family in the Family list.
8. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard® Plug-In Manager by clicking **MegaWizard Plug-In Manager** Tools menu). The MegaWizard Plug-In Manager dialog box displays.


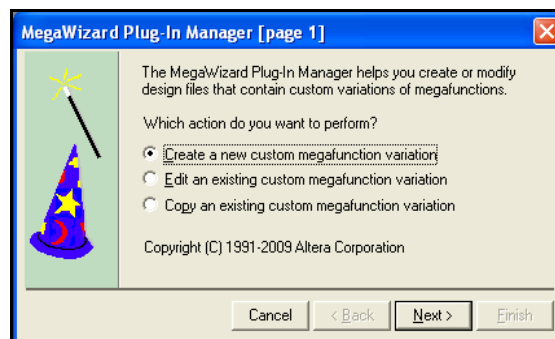
 Refer to the Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

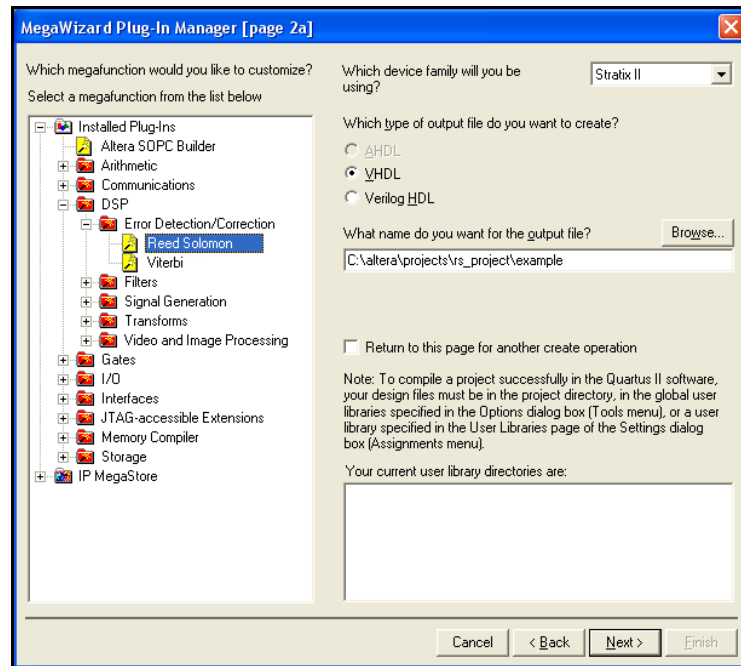
Figure 2–2. MegaWizard Plug-In Manager



2. Specify that you want to create a new custom megafunction variation and click **Next**.

3. Expand the **DSP > Error Detection/Correction** directory then click **Reed-Solomon**.
4. Choose the output file type for your design; the wizard supports VHDL and Verilog HDL.
5. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files `<project path>\<variation name>`. [Figure 2-3](#) shows the wizard after you have made these settings.

Figure 2-3. Select the Megafunction



6. Click **Next** to launch IP Toolbench.

Step 1: Parameterize

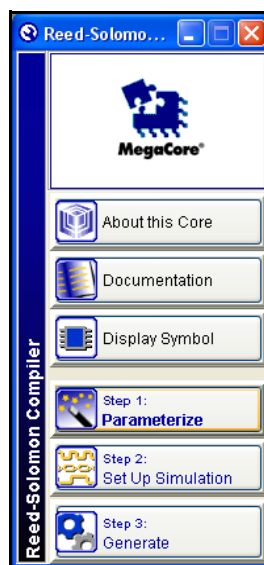
To parameterize your MegaCore function, follow these steps:



For more information on the parameters, refer to [“Parameters” on page 3-8](#).

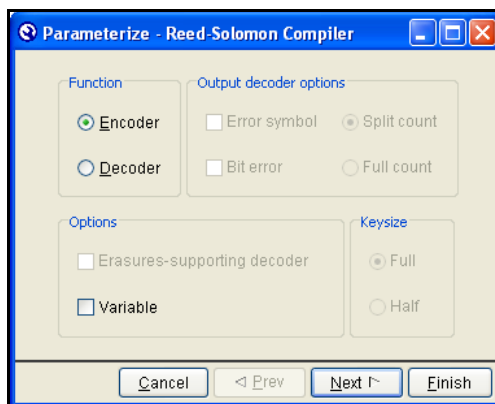
1. Click **Step 1: Parameterize** in IP Toolbench (see [Figure 2-4](#)).

Figure 2-4. IP Toolbench—Parameterize




2. Select **Encoder** or **Decoder** (see [Figure 2-5](#)).


Figure 2-5. Select the Encoder or Decoder



3. For the encoder you can turn on the **Variable** option.

 For more information on the variable option, see [“Variable Encoding & Decoding”](#) on page 3-3.

4. For the decoder:
 - a. You can turn on the **Erasures-supporting** or **Variable** options.
 - b. Select **Full** or **Half** keysize.
 - c. You can turn on the **Error Symbol** or **Bit Error** outputs. For the bit error output, select **Split Count** or **Full Count**.

 For more information on the parameters, see [Table 3-2](#) on page 3-8.

5. Click **Next**.
6. Select the parameters that define the specific RS codeword that you wish to implement (see [Figure 2-6](#)). You can enter the parameters one by one, or click **DVB Standard** to use digital video broadcast (DVB) standard values, or **CCSDS Standard** to use the CCSDS standard values.


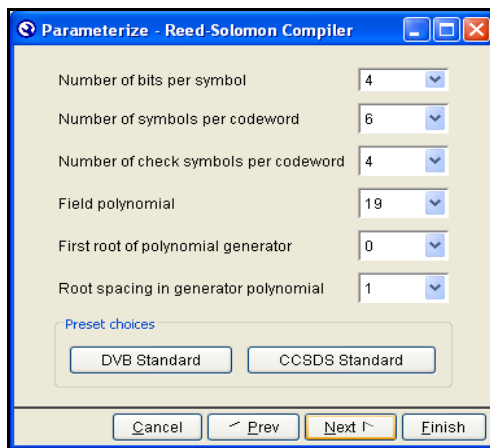
 For more information on the parameters, see [Table 3-3 on page 3-8](#).

Figure 2-6. Choose the Parameters



7. Click **Next**.
8. For a decoder throughput calculation, enter the frequency in MHz, select the desired units, and click **Calculate** (see [Figure 2-7](#)).


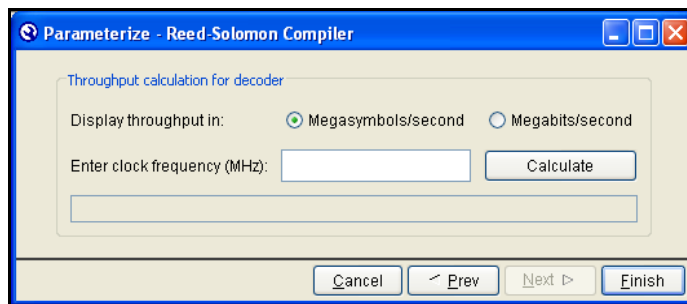
 For more information on the throughput calculator, see [“Throughput Calculator” on page 3-11](#).

Figure 2-7. Throughput Calculator



9. Click **Finish**.
- To view the symbol, click **Display Symbol** in IP Toolbench.

Step 2: Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

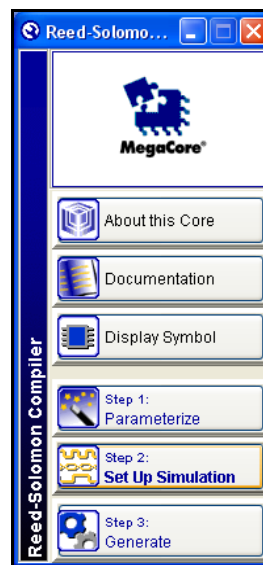


You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

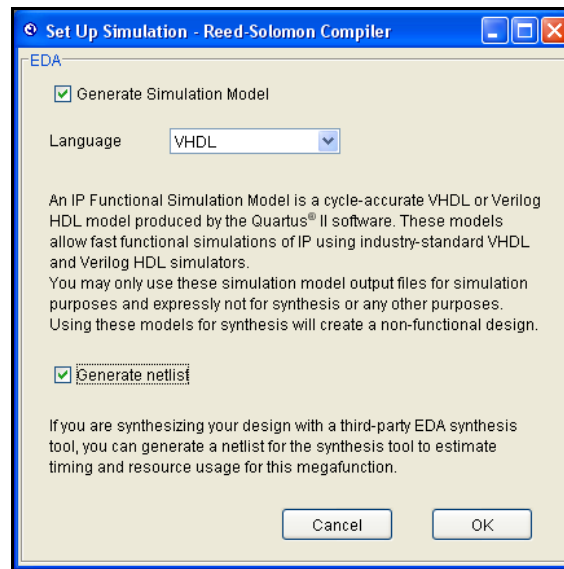
To generate an IP functional simulation model for your MegaCore function, follow these steps:

1. Click **Step 2: Set Up Simulation** in IP Toolbench (see [Figure 2-8](#)).

Figure 2-8. Set Up Simulation



2. Turn on **Generate Simulation Model** (see [Figure 2-9](#)).

Figure 2–9. Generate Simulation Model

3. Choose the language in the **Language** list.



Choose the same language as your design.

4. Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.
5. Click **OK**.

Step 3: Generate

To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench (see [Figure 2–10](#)).

Figure 2-10. IP Toolbench—Generate



Figure 2-11 shows the generation report.

Figure 2-11. Generation Report

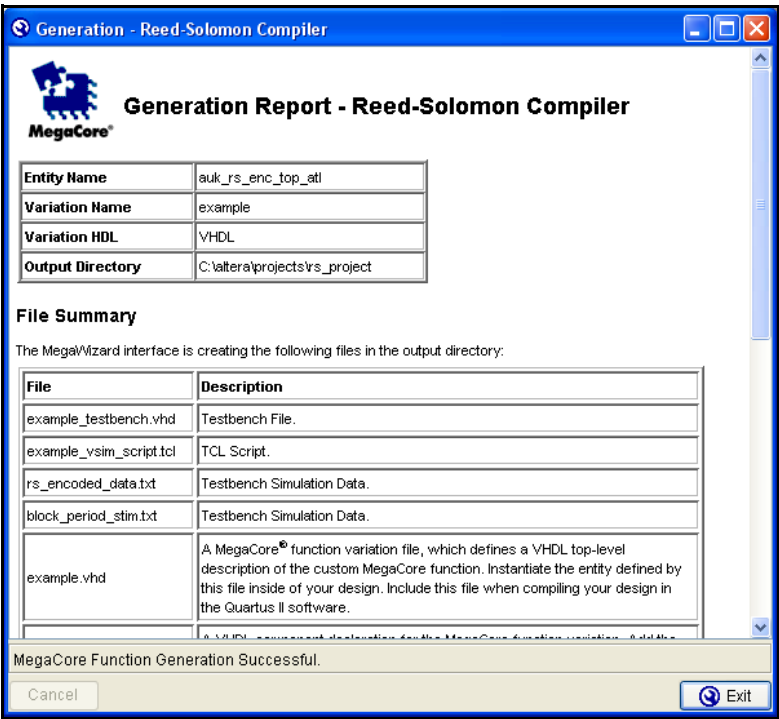


Table 2-1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

Table 2-1. Generated Files (Note 1)

Filename	Description
<variation name>.bsf	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<variation name>.vo or .vho	VHDL or Verilog HDL IP functional simulation model.
<variation name>.vhd, or .v	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<variation name>_nativelink.tcl	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools.
<variation name>_syn.v or <variation name>_syn.vhd	A timing and resource netlist for use in some third-party synthesis tools.
<variation name>_testbench.vhd	The testbench variation file, which defines the top-level testbench that runs the simulation. This file instantiates the function variation file and the testbench from the reed_solomon\lib directory.
<variation name>_vsim_script.tcl	Starts the MegaCore function simulation in the ModelSim simulator.
block_period_stim.txt	The testbench stimuli includes information such as number of codewords, number of symbols, and check symbols for each codeword
rs_encoded_data.txt	Contains the encoded test data.

Notes to Table 2-1:

(1) <variation name> is the variation name.

- After you review the generation report, click **Exit** to close IP Toolbench and click **Yes** on the **Quartus II IP Files** message.




The Quartus II IP File (.qip) is a file generated by the MegaWizard interface that contains information about a generated IP core. You are prompted to add this .qip file to the current Quartus II project at the time of file generation. In most cases, the .qip file contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single .qip file is generated for each MegaCore function.

You can now integrate your custom variation into your design and simulate and compile.

Simulate the Design

IP Toolbench-generated Tcl scripts drive the simulation. For the decoder, the testbench includes a channel and the instantiated decoder. Data is read from an IP Toolbench-generated file. For the encoder, the testbench reads the same data file and just compares the encoder output with a data file. In the channel, some errors are introduced at various locations of the RS codeword. The testbench then receives the data decoded by the RS decoder and compares it with the originally transmitted data.

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.

 For more information on NativeLink, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

You can use the Tcl script file `<variation name>_nativelink.tcl` to assign default NativeLink testbench settings to the Quartus II project.

To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation but ensure you specify your variation name to match the Quartus II project name.
2. Check that the absolute path to your third-party simulator executable is set. On the Tools menu click **Options** and select **EDA Tools Options**.
3. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
4. On the Tools menu click **Tcl scripts**. Select the `<variation name>_nativelink.tcl` Tcl script and click **Run**. Check for a message confirming that the Tcl script was successfully loaded.
5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name**.
6. On the Tools menu point to **EDA Simulation Tool** and click **EDA RTL Simulation**.


Compile the Design

You can use the Quartus II software to compile your design. Refer to Quartus II Help for instructions on performing compilation.


Program a Device

After you have compiled your design, program your targeted Altera device and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate an RS MegaCore function before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model and produce a time-limited programming file.

 For more information on IP functional simulation models, see the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

You can simulate an RS MegaCore function in your design and perform a time-limited evaluation of your design in hardware.

 For more information on OpenCore Plus hardware evaluation using the RS Compiler, see “*OpenCore Plus Evaluation*” on page 1-3, “*OpenCore Plus Time-Out Behavior*” on page 3-7, and AN 320: *OpenCore Plus Evaluation of Megafunctions*.

Set Up Licensing

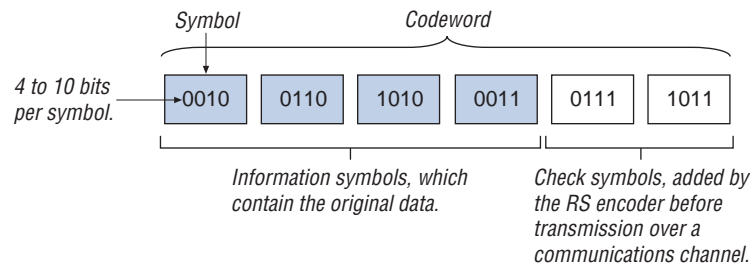
You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance and want to take your design to production.

After you purchase a license for RS Compiler, you can request a license file from the Altera web site at www.altera.com/licensing and install it on your computer. When you request a license file, Altera emails you a license.dat file. If you do not have Internet access, contact your local Altera representative.

Background

To use Reed-Solomon (RS) codes, a data stream is first broken into a series of codewords. Each codeword consists of several information symbols followed by several check symbols (also known as parity symbols or redundant symbols). Symbols can contain an arbitrary number of bits. In an error correction system, the encoder adds check symbols to the data stream prior to its transmission over a communications channel. When the data is received, the decoder checks for and corrects any errors (see [Figure 3-1](#)).

Figure 3-1. RS Codeword Example



RS codes are described as (N,K) , where N is the total number of symbols per codeword and K is the number of information symbols. R is the number of check symbols ($N - K$). Errors are defined on a symbol basis. Any number of bit errors within a symbol is considered as only one error.

RS codes are based on finite-field (i.e., Galois field) arithmetic. Any arithmetic operation (addition, subtraction, multiplication, and division) on a field element gives a result that is an element of the field. The size of the Galois field is determined by the number of bits per symbol—specifically, the field has 2^m elements, where m is the number of bits per symbol. A specific Galois field is defined by a polynomial, which is user-defined for the RS Compiler. IP Toolbench lets you select only valid field polynomials.

The maximum number of symbols in a codeword is limited by the size of the finite field to $2^m - 1$. For example, a code based on 10-bit symbols can have up to 1,023 symbols per codeword. The RS Compiler supports shortened codewords.

The following equation represents the generator polynomial of the code:

$$g(x) = \prod_{i=0}^{R-1} (x - \alpha^{a \cdot i + i_0})$$

where:

- i_0 is the first root of the generator polynomial
- a is the rootspace
- R is the number of check symbols
- α is a root of the polynomial.

For example, for the following information:

$$g(x) = \prod_{i=0}^3 (x - \alpha^{i+i_0})$$

a is a root of the binary primitive polynomial $x^8 + x^7 + x^2 + x + 1$
 $i_0 = 120$

You can calculate the following parameters:

- $R - 1 = 3$
- $a = 1$ (α is to the power 1 times i)

The field polynomial can be obtained by replacing x with 2, thus:

$$2^8 + 2^7 + 2^2 + 2 + 1 = 391$$

Erasures

In normal operation the RS decoder detects and corrects symbol errors.

The number of symbol errors that can be corrected, C , depends on the number of check symbols, R and is given by $C \leq R/2$.

If the location of the symbol errors is marked as an erasure, the RS decoder can correct twice as many errors, so $C \leq R$.



Erasures are symbol errors with a known location.

External circuitry identifies which symbols have errors and passes this information to the decoder using the `eras_sym` signal. The `eras_sym` input indicates an erasure (when the erasures-supporting decoder option is selected).

The RS decoder can work with a mixture of erasures and errors.

A codeword is correctly decoded if $(2e + E) \leq R$

where:

- e = errors with unknown locations
- E = erasures
- R = number of check symbols.

For example, with ten check symbols the decoder can correct ten erasures, or five symbol errors, or four erasures and three symbol errors.



If the number of erasures marked approaches the number of check symbols, the ability to detect errors without correction (`decfail` asserted) diminishes (see [Table 3-1 on page 3-4](#)).

Shortened Codewords

A shortened codeword contains fewer symbols than the maximum value of N , which is $2^m - 1$. A shortened codeword is mathematically equivalent to a maximum-length code with the extra data symbols at the start of the codeword set to 0.

For example, (204,188) is a shortened codeword of (255,239). Both of these codewords use the same number of check symbols, 16.

To use shortened codewords with the Altera RS encoder and decoder, you use IP Toolbench to set the codeword length to the correct value, in the example, 204.

Variable Encoding & Decoding

Under normal circumstances, the encoder and decoder allow variable encoding and decoding—you can change the number of symbols per codeword (N) using `sink_eop`, but not the number of check symbols while decoding.



However, you cannot change the length of the codeword, if you turn on the erasure-supporting option.

If you turn on the variable option, you can vary the number of symbols per codeword (using the `numn` signal) and the number of check symbols (using the `numcheck` signal), in real time, from their minimum allowable values up to their selected values, even with the erasures-supporting option turned on. [Table 3-7 on page 3-10](#) shows the variable option signals.

RS Encoder

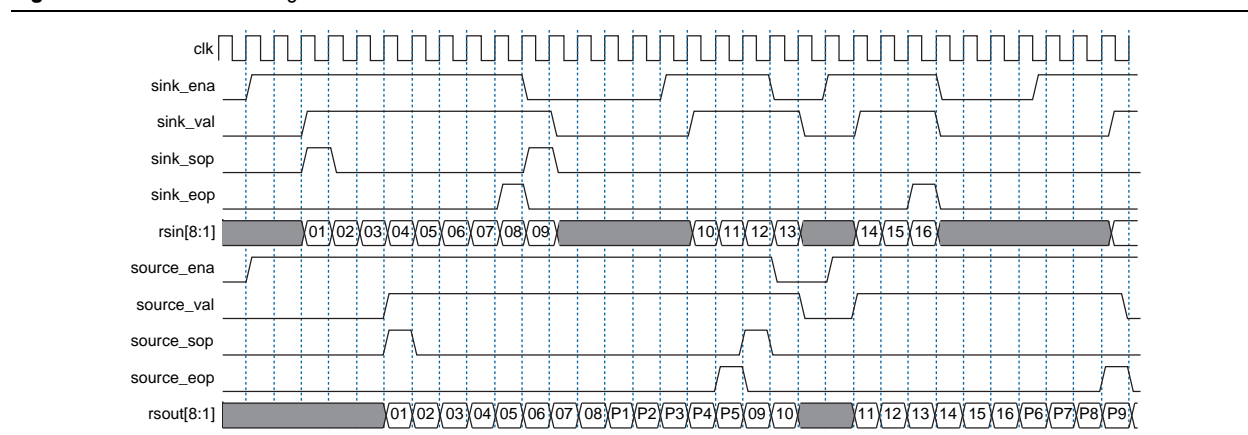
The `sink_sop` signal starts a codeword; `sink_eop` signals its termination. An asserted `sink_val` indicates valid data. The `sink_sop` is only valid when `sink_val` is asserted.



Only assert `sink_val` one clock cycle after the encoder asserts `sink_ena`.

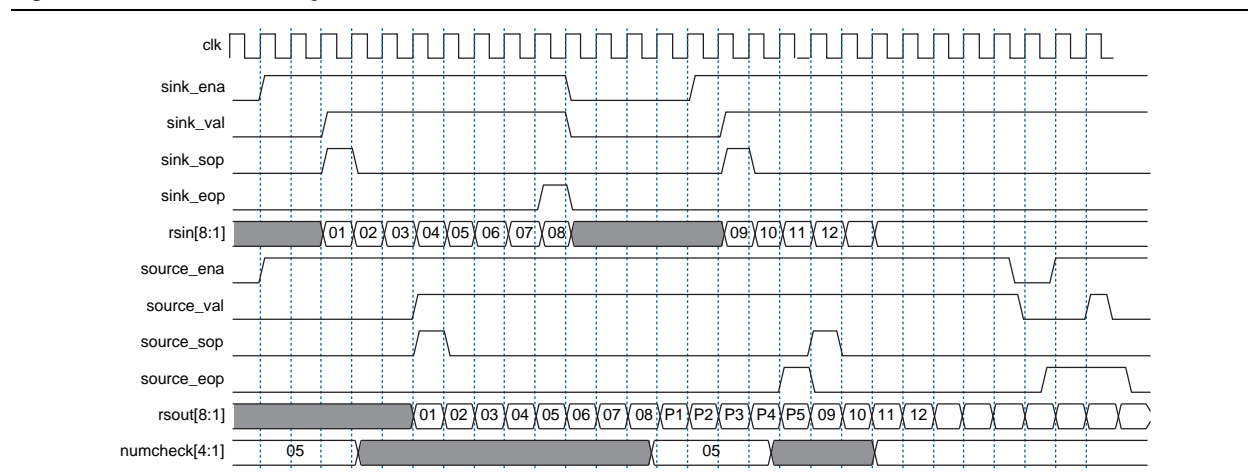
By de-asserting `sink_ena`, the encoder signals that it cannot sink more incoming symbols after `sink_eop` is signalled at the input. During this time it is generating the check symbols for the current codeword. [Figure 3-2](#) shows the operation of the RS encoder. The example shows a codeword with eight information symbols and five check symbols.

Figure 3-2. Encoder Timing



The `numcheck` input is latched inside the encoder when `sink_sop` is asserted.

You can change the number of symbols in a codeword at run-time without resetting the encoder. You must make the changes between complete codewords; you cannot change `numcheck` during encoding. [Figure 3-3](#) shows variable encoding.

Figure 3-3. Variable Encoding

RS Decoder

The decoder implements an Avalon-ST-based pipelined three-codeword-depth architecture. However, if the parameters are in the continuous range (see [Table 3-3](#)), the decoder shows continuous behavior and can accept a new symbol every clock cycle.

The decoder is self-flushing—it processes and delivers a codeword without needing a new codeword to be fed in. Therefore, latency between the input and output does not depend on the availability of input data. The throughput latency is approximately three codewords.

The reset is active high and can be asserted asynchronously. However, it has to be de-asserted synchronously with `clk`.

The RS decoder always tries to detect and correct errors in the codeword. However, as the number of errors increases, the decoder gets to a stage where it can no longer correct but only detect errors, at which point the decoder asserts the `decfail` signal. As the number of errors increases still further, the results become unpredictable. [Table 3-1](#) shows how the decoder corrects and detects errors depending on R .

Table 3-1. Decoder Detection and Correction

Number of Errors	Decoder Behavior
$\text{Errors} \leq R/2$	Decoder detects and corrects errors.
$R/2 \leq \text{errors} \leq R$	Decoder asserts <code>decfail</code> and can only detect errors. (1)
$\text{Errors} > R$	Unpredictable results.

Note to [Table 3-1](#):

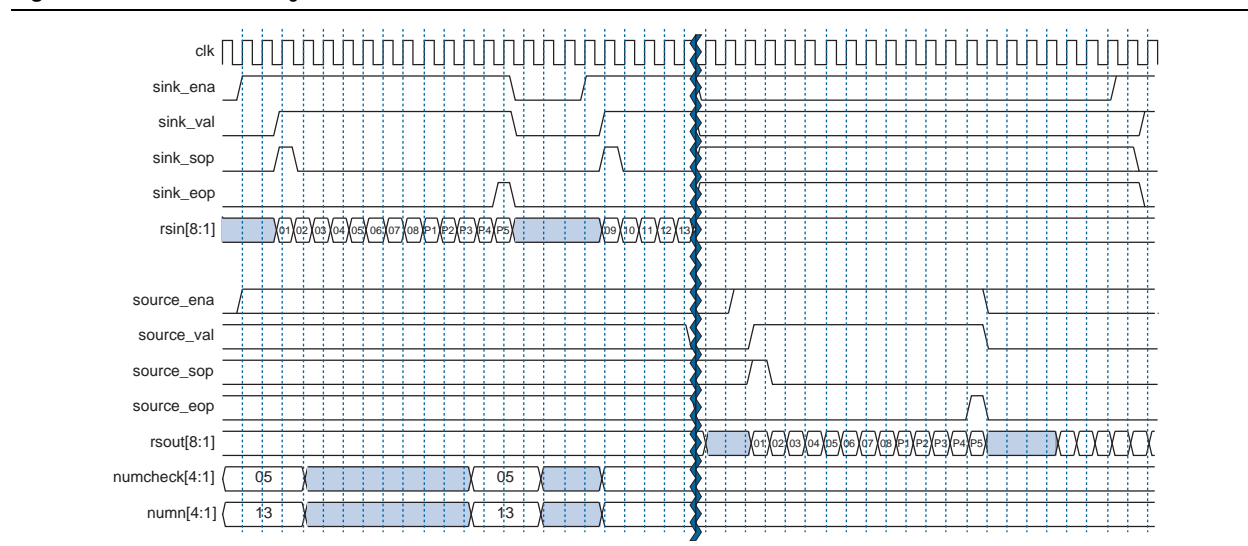
(1) The decoder may fail to assert `decfail`, for low values of R (4, 5, or 6), or when using erasures and the differences between the number of erasures and R is small (4, 5 or 6).

The RS decoder observes Avalon-ST interface standard for input and output data. One clock cycle after the decoder asserts `sink_ena`, you can assert `sink_val`. The decoder accepts the data at `rsin` as valid data. The codeword is started with `sink_sop`. The `numcheck` and `numn` signals are latched to `sink_sop`. The codeword is finished when `sink_eop` is asserted. If `sink_ena` is de-asserted, from one clock cycle onwards the decoder cannot process any more data until `sink_ena` is asserted again.

At the output the operation is identical. If you assert `source_ena`, the decoder asserts `source_val` and provides valid data on `rsout` if available. Also, it indicates the start and end of the codeword with `source_sop` and `source_eop` respectively.

Figure 3-4 shows the operation of the RS decoder.

Figure 3-4. Decoder Timing

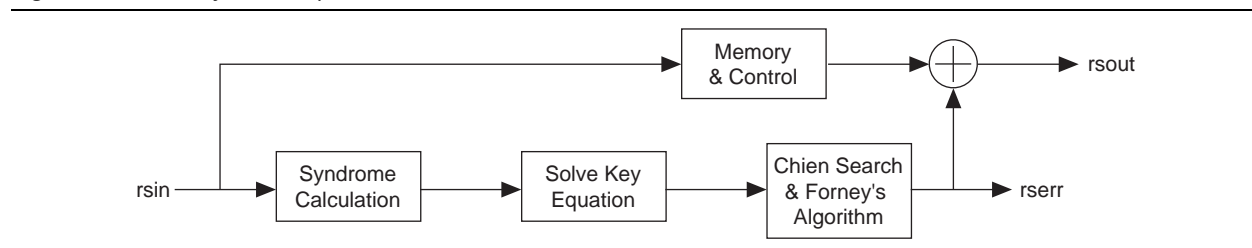


The decoder has the following optional outputs, which you turn on in IP Toolbench:

- Error symbol
- Bit error count

Error Symbol Output

The error symbol output, `rserr` is the Galois field error correction value. The RS decoder finds the error values and location and adds these values in the Galois field to the input value. Galois field addition and subtraction is the same operation. An XOR operation performs this operation between bits of the two values. Figure 3-5 shows the error symbol output.

Figure 3-5. Error Symbol Output

Whenever `rserr` is not 0 (while `decfail` is 0), an error correction successfully takes place. The `rsout` is the `rserr` XORed with the corresponding `rsin`, where XOR is done for each bit, so you know that the respective symbol has been corrected. The value of `rserr` shows which bits of the symbol have been corrected. For each bit of `rserr` that is 1, the corresponding bit of `rsout` is corrected.

The `rsout` and the corresponding `rserr` value appear at the output at the same clock cycle.

Bit Error Count

The decoder can provide the bit error count found in the correction process. The bit error count has the following options:

- Full count. The output `num_err_bit` is connected, which shows the valid value.
- Split count. The outputs `num_err_bit0` and `num_err_bit1` are connected, which show the valid values



For information on these outputs, see [Table 3-8 on page 3-10](#).

Interfaces

The RS encoder and decoder use the Avalon® Streaming (Avalon-ST) interface for data input and output. The input is an Avalon-ST sink and the output is an Avalon-ST source. The Avalon-ST interface `READY_LATENCY` parameter is set to 1. The Avalon-ST interfaces allow for flow control.

The Avalon-ST interface is an evolution of the Atlantic™ interface. The Avalon-ST interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals. The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism, where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output. When designing a datapath, which includes the RS MegaCore function, you may not need backpressure if you know the downstream components can always receive data. You may achieve a higher clock rate by driving the source ready signal `source_ena` of the RS high, and not connecting the sink ready signal `sink_ena`.


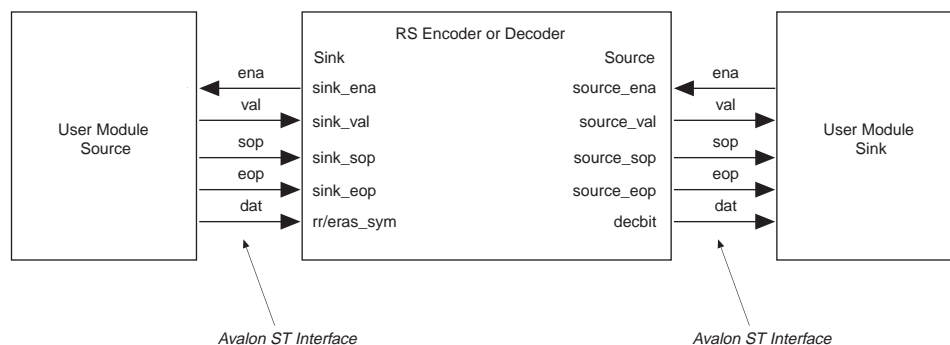
 For more information on the Avalon-ST interface, refer to the [Avalon Interface Specification](#).

Figure 3-6 shows the RS encoder and decoder Avalon-ST interfaces.

Figure 3-6. Avalon ST Interface




OpenCore Plus Time-Out Behavior


OpenCore Plus hardware evaluation can support the following two modes of operation:

- Untethered—the design runs for a limited time.
- Tethered—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.

 For MegaCore functions, the untethered timeout is 1 hour; the tethered timeout value is indefinite.

Your design stops working after the hardware evaluation time expires and the data output `rsout` remains low.

 For more information on OpenCore Plus hardware evaluation, see [“OpenCore Plus Evaluation” on page 1-3](#) and [AN 320: OpenCore Plus Evaluation of Megafunctions](#).

Parameters

Table 3-2 shows the implementation parameters.

Table 3-2. Implementation Parameters

Parameter	Value	Description
Function	Encoder or decoder.	Specifies an encoder or a decoder (see “Functional Description” on page 3-1).
Variable option	On or off.	Specifies the variable option (see “Variable Encoding & Decoding” on page 3-3).
Erasures-supporting option	On or off.	Specifies the erasures-supporting option. Erasures-supporting substantially increases the logic resources used (see “Erasures” on page 3-2).
Error symbol output (1)	On or off.	Specifies the error symbol output (see “RS Decoder” on page 3-4 and Table 3-8 on page 3-10).
Bit error output (1)	On or off.	Specifies the bit error output as either split or full count (see “RS Decoder” on page 3-4 and Table 3-8 on page 3-10).
Keysize (1)	Half or full.	The keysize parameter allows you to trade off the amount of logic resources against the supported throughput. Full has twice as many Galois field multipliers as half. A full decoder uses more logic and is probably slightly slower in frequency, but supports a higher throughput. If both full and half give you the required throughput for your parameters, always select half.

Note to Table 3-2:

(1) This parameter applies to the decoder only.

Table 3-3 shows the RS codeword parameters.

Table 3-3. RS Codeword Parameters

Parameter	Range	Range (Continuous)	Description
Number of bits per symbol (m)	3 to 12	6 to 12	Specifies the number of bits per symbol.
Number of symbols per codeword (N)	5 to $(2^m - 1)$	$7(R + 1)$ to $2^m - 1$	Specifies the total number of symbols per codeword.
Number of check symbols per codeword (R)	2 to $\min(128, N - 1)$	4 to $N/7 - 1$	Specifies the number of check symbols per codeword.
Field polynomial	Any valid polynomial (1)		Specifies the primitive polynomial defining the Galois field.
First root of generator polynomial (i_0)	0 to $(2^m - 2)$		Specifies the first root of the generator polynomial.
Root spacing in generator polynomial (a)	Any valid root space (1)		Specifies the minimum distance between roots in the generator polynomial.

Notes to Table 3-3:

(1) Toolbench allows you to select only legal values. For $m > 8$, not all legal values of the field polynomials and rootspace are present in IP Toolbench. If you cannot find your intended field polynomial or rootspace in the IP Toolbench list, contact Altera MySupport.

Signals

Table 3-4 shows the global signals.

Table 3-4. Global Signals

Name	Description
clk	clk is the main system clock. The whole MegaCore function operates on the rising edge of clk.
reset	Reset. The entire decoder is asynchronously reset when reset is asserted high. The reset signal resets the entire system. The reset signal must be de-asserted synchronously with respect to the rising edge of clk.

Table 3-5 shows the Avalon-ST sink (data input) interface.

Table 3-5. Avalon-ST Sink Interface

Name	Avalon-ST Type	Direction	Description
sink_ena	ena	Output	Data transfer enable signal. sink_ena is driven by the sink interface and controls the flow of data across the interface. sink_ena behaves as a read enable from sink to source. When the source observes sink_ena asserted on the clk rising edge it drives, on the following clk rising edge, the Avalon-ST data interface signals and asserts val, if data is available. The sink interface captures the data interface signals on the following clk rising edge. If the source is unable to provide new data, it de-asserts val for one or more clock cycles until it is prepared to drive valid data interface signals.
sink_val	val	Input	Data valid signal. sink_val indicates the validity of the data signals. sink_val is updated on every clock edge where sink_ena is asserted. sink_val and the dat bus hold their current value if sink_ena is de-asserted. When sink_val is asserted, the Avalon-ST data interface signals are valid. When sink_val is de-asserted, the Avalon-ST data interface signals are invalid and must be disregarded. To determine whether new data has been received, the sink interface qualifies the sink_val signal with the previous state of the sink_ena signal.
sink_sop	sop	Input	Start of packet (codeword) signal. sop delineates the codeword boundaries on the rsin bus. When sink_sop is high, the start of the packet is present on the rsin bus. sink_sop is asserted on the first transfer of every codeword.
sink_eop	eop	Input	End of packet (codeword) signal. sink_eop delineates the packet boundaries on the rsin bus. When sink_eop is high, the end of the packet is present on the dat bus. sink_eop is asserted on the last transfer of every packet.
rsin[m:1]	dat	Input	Data input for each codeword, symbol by symbol. Valid only when sink_val is asserted.
eras_sym	dat	Input	When asserted, the symbol in rsin[] is marked as an erasure. Valid only for the decoder with erasures-supporting option.

Table 3-6 shows the Avalon-ST source (data output) interface.

Table 3-6. Avalon-ST Source Interface

Name	Avalon-ST Type	Direction	Description
source_ena	ena	Input	Data transfer enable signal. <code>source_ena</code> is driven by the sink interface and controls the flow of data across the interface. <code>ena</code> behaves as a read enable from sink to source. When the source interface observes <code>source_ena</code> asserted on the <code>clk</code> rising edge it drives, on the following <code>clk</code> rising edge, the Avalon-ST data interface signals and asserts <code>source_val</code> when data from sink interface is available. The sink interface captures the data interface signals on the following <code>clk</code> rising edge. If this source is unable to provide new data, it de-asserts <code>source_val</code> for one or more clock cycles until it is prepared to drive valid data interface signals.
source_val	val	Output	Data valid signal. <code>source_val</code> is asserted high, whenever there is a valid output on <code>rsout</code> ; it is deasserted when there is no valid output on <code>rsout</code> .
source_sop	sop	Output	Start of packet (codeword) signal.
source_eop	eop	Output	End of packet (codeword) signal.
rsout	dat	Output	The <code>rsout</code> signal contains decoded output when <code>source_val</code> is asserted. The corrected symbols are in the same order that they were entered.
rserr	dat	Output	Error correction value (decoder only, optional), see “Error Symbol Output” on page 3-5.

Table 3-7 shows the configuration signals.

Table 3-7. Configuration Signals

Name	Description
bypass	A one-bit signal that sets if the codewords are bypassed or not (decoder only). The decoder continuously samples <code>bypass</code> .
numcheck	Sets the variable number of check symbols up to a maximum value set by the parameter <i>R</i> (variable option only). The decoder samples <code>numcheck</code> only when <code>sink_sop</code> is asserted.
numn	Variable value of <i>N</i> . Can be any value from the minimum allowable value of <i>N</i> up to the selected value of <i>N</i> (variable and erasures-supporting option only). The decoder samples <code>numn</code> only when <code>sink_sop</code> is asserted.

Table 3-8 shows the status signals (decoder only).

Table 3-8. Status Signals (Part 1 of 2)

Name	Description
decfail	Indicates non-correctable codeword. Valid when <code>source_sop</code> is asserted. Avalon-ST type <code>err</code> .
num_err_sym	Number of symbols errors. Valid when <code>source_sop</code> is asserted; invalid when <code>decfail</code> is asserted.
num_err_bit	Number of bits errors corrected in the codeword. Valid when <code>source_sop</code> is asserted; invalid when <code>decfail</code> is asserted. Connected only when bit error (full count) option is turned on, see “RS Decoder” on page 3-4.

Table 3–8. Status Signals (Part 2 of 2)

Name	Description
num_err_bit0	Number of bit errors for the corrections from bit 1 to bit 0. The latest is the correct bit. Valid when <code>sop_source</code> is asserted; invalid when <code>decfail</code> is asserted. The decoder presents these values at the next <code>source_sop</code> assertion (at the next codeword). Connected only when bit error (split count) option is turned on.
num_err_bit1	Number of bit errors for the corrections from bit 0 to bit 1. The latest is the correct bit. Valid when <code>sop_source</code> is asserted; invalid when <code>decfail</code> is asserted. The decoder presents these values at the next <code>source_sop</code> assertion (at the next codeword). Connected only when bit error (split count) option is turned on.

Throughput Calculator

The IP Toolbench throughput calculator (decoder only) uses the following equation:

Throughput in megasymbols per second = $N \times \text{frequency (MHz)} / N_C$

For Mbps, multiply by m , the number of bits per symbol.

Table 3–9 shows the value of N_C .

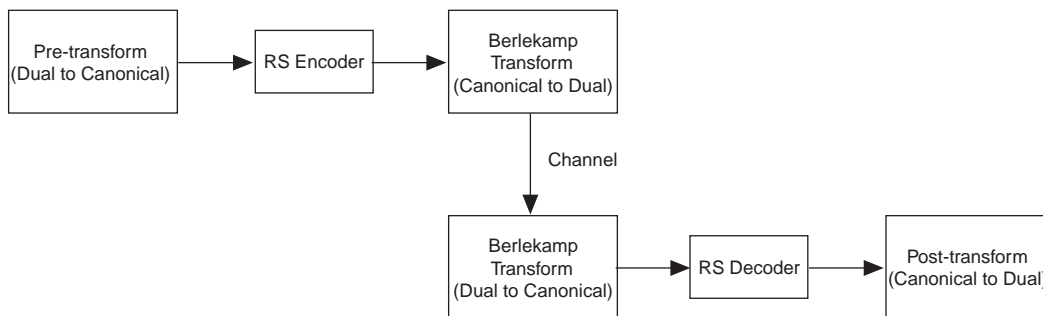
Table 3–9. Calculate N_C

Erasures	Keysize	N_C
No	Half	$\text{Max}(N, 10 \times R + 4)$
No	Full	$\text{Max}(N, 7 \times R + 5)$
Yes	Half	$\text{Max}(N, 10 \times R + 6)$
Yes	Full	$\text{Max}(N, 8 \times R + 4)$

Introduction

The Reed-Solomon (RS) encoder or decoder MegaCore® functions work in canonical base (otherwise known as conventional base). This base can cause confusion when trying to implement the RS encoder or decoder directly into a dual-base system, for example, when working with the CCSDS standard. To transfer from a canonical-base to a dual-base system, a Berlekamp transform is used, which you need to implement in logic. [Figure A-1](#) shows an example use of the Berlekamp transform.

Figure A-1. Using the Berlekamp Transform



Test Patterns

If you are working with a dual-base system, e.g., CCSDS, and wish to supply the RS encoder or decoder with some test patterns from the dual-base system, follow these steps:

1. Apply the Berlekamp transform (dual to canonical) to the test pattern.
2. Apply the test pattern to RS encoder or decoder.
3. Apply the Berlekamp transform (canonical to dual) to the encoder output.
4. Check the test pattern.



For more information on how to implement the transformation function, see *Annex B* of the standard specification document *CCSDS-101.0-B-5* at www.ccsds.org.

Revision History

The following table shows the revision history for this user guide.

Date	Version	Changes Made
March 2009	9.0	Added Arria® II GX device support
November 2008	8.1	No changes.
May 2008	8.0	Added device support for Stratix® IV devices.
October 2007	7.2	No changes.
May 2007	7.1	Updated <code>rserr</code> signal
December 2006	7.0	Added support for Cyclone® III devices.
December 2006	6.1	Updated format.

How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.

Contact <i>(Note 1)</i>	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General) (Software Licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com






Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, \qdesigns directory, d: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicates document titles. For example: <i>AN 519: Stratix IV Design Guidelines</i> .

Visual Cue	Meaning
<i>Italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>. .pcf file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, data1, tdi, and input. Active-low signals are denoted by suffix n. Example: resetn. Indicates command line commands and anything that must be typed exactly as it appears. For example, c:\qdesigns\tutorial\chiptrip.gdf. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword SUBDESIGN), and logic function names (for example, TRI).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press the enter key.
	The feet direct you to more information about a particular topic.