

Reed-Solomon Compiler

User Guide



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com

Compiler Version:	4.1.0
Document Version:	4.1.0
Document Date:	April 2006

Copyright © 2006 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.



I.S. EN ISO 9001

UG-RSCOMPILER-4.3

About This User Guide	v
Revision History	v
How to Contact Altera	vii
Typographic Conventions	viii
Chapter 1. About This Compiler	
Release Information	1-1
Device Family Support	1-1
New in Version 4.1.0	1-2
Features	1-2
General Description	1-2
OpenCore Plus Evaluation	1-3
DSP Builder Support	1-3
Performance	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	2-9
Simulate the Design	2-11
Compile the Design	2-12
Program a Device	2-12
Set Up Licensing	2-12
Chapter 3. Specifications	
Functional Description	3-1
Erasures	3-2
Shortened Codewords	3-3
Variable Encoding & Decoding	3-3
Interfaces	3-4
RS Encoder	3-4
RS Decoder	3-6
OpenCore Plus Time-Out Behavior	3-9
Parameters	3-9
Signals	3-11
MegaCore Verification	3-14

Contents

Throughput Calculator	3-14
Appendix A: Using the RS Encoder or Decoder in a CCSDS System	
Introduction	A-1
Test Patterns	A-1
Appendix B: Implementing Atlantic Slave Source Interface Control	
Stimulus & Response Block Files	B-4
Appendix C: Information for Version 3.6.0 Users	



About This User Guide

Revision History The table below displays the revision history for the chapters in this user guide.

Chapter	Date	Version	Changes Made
All	April 2006	4.1.0	Implemented minor format changes.
1	November 2005	4.0.1	No changes.
	October 2005	4.0.0	<ul style="list-style-type: none">• Updated device family support table.• Updated performance table.
	June 2004	3.5.0	<ul style="list-style-type: none">• Updated device family support table.• Added DSP Builder support.
	February 2004	3.4.0	<ul style="list-style-type: none">• Updated device family support table.• Added OpenCore® Plus description.• Updated performance figures.
2	November 2005	4.0.1	No changes.
	October 2005	4.0.0	<ul style="list-style-type: none">• Updated flow.• Updated simulation walkthrough.• Updated generated files table.
	June 2004	3.5.0	<ul style="list-style-type: none">• Updated system requirements.• Updated Quartus® II software information.
	February 2004	3.4.0	<ul style="list-style-type: none">• Added IP Toolbench flow.• Added IP functional simulation models information.
3	November 2005	4.0.1	No changes.
	October 2005	4.0.0	<ul style="list-style-type: none">• Rewritten encoder and decoder descriptions for Atlantic™ interfaces.• Rewritten signals tables.• Updated parameters table.• Updated encoder timing figures.• Updated <code>reset</code> description.
	July 2004	3.6.0	Updated <code>reset</code> signal information.
	February 2004	3.4.0	Added OpenCore Plus information.
A	November 2005	4.0.1	No changes.
	October 2005	4.0.0	No changes.
	February 2004	3.4.0	Added Appendix A.
B	November 2005	4.0.1	No changes.
	October 2005	4.0.0	Added Appendix B.

Revision History

Chapter	Date	Version	Changes Made
C	November 2005	4.0.1	No changes.
	October 2005	4.0.0	Added Appendix C.








How to Contact Altera

For the most up-to-date information about Altera® products, go to the Altera world-wide web site at www.altera.com. For technical support on this product, go to www.altera.com/mysupport. For additional information about Altera products, consult the sources shown below.

Information Type	USA & Canada	All Other Locations
Technical support	www.altera.com/mysupport/	www.altera.com/mysupport/
	800-800-EPLD (3753) 7:00 a.m. to 5:00 p.m. Pacific Time	+1 408-544-8767 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
Product literature	www.altera.com	www.altera.com
Altera literature services	literature@altera.com	literature@altera.com
Non-technical customer service	800-767-3753	+ 1 408-544-7000 7:00 a.m. to 5:00 p.m. (GMT -8:00) Pacific Time
FTP site	ftp.altera.com	ftp.altera.com

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: Save As dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: f_{MAX} , qdesigns directory, d: drive, chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t_{PIA}</i> , <i>n + 1</i> . Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <file name>, <project name>.pdf file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn. Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword SUBDESIGN), as well as logic function names (e.g., TRI) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets are used in a list of items when the sequence of the items is not important.
	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	The caution indicates required information that needs special consideration and understanding and should be read prior to starting or continuing with the procedure or process.
	The warning indicates information that should be read prior to starting or continuing the procedure or processes
	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.



1. About This Compiler

Release Information

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

<i>Table 1–1. RS Compiler Release Information</i>	
Item	Description
Version	4.1.0
Release Date	April 2006
Ordering Codes	IP-RSENC (Encoder) IP-RSDEC (Decoder)
Product IDs	0039 0041 (Encoder) 0080 0041 (Decoder)
Vendor ID	6AF7

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.

<i>Table 1–2. Device Family Support (Part 1 of 2)</i>	
Device Family	Support
Stratix® II	Full
Stratix II GX	Preliminary
Stratix GX	Full
Stratix	Full
Cyclone™ II	Full

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

Device Family	Support
Cyclone	Full
HardCopy® II	Preliminary
HardCopy Stratix	Full

New in Version 4.1.0

- Maintenance release

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 3-9](#) 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 generate hardware implementations. 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. To use this MegaCore function with DSP Builder, you require DSP Builder v6.0 or higher and the Quartus® II software version 6.0 or higher.



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

Table 1–3 shows the typical performance using the Quartus II software version 6.0 for the following devices:

- Stratix II EP2S15F484C3
- Cyclone II EP2C5T144C6

The throughput in megabits per second (Mbps) is derived from the formulas in Table 3–9 on page 3–15 and maximum frequency at which the design can operate.

Table 1–3. Performance									
Device	Options	Keysize	m	N	check	LEs or ALUTs (1)	Memory (M4K)	f_{MAX} (MHz)	Throughput (Mbps)
Stratix II	None	half	8	204	16	1,476	7	250	2,000
	Variable	half	8	204	16	1,644	7	241	1,933
	Erasures	half	8	204	16	2,719	7	224	1,792
	Variable and erasures	half	8	204	16	2,912	8	221	1,773
	None	half	8	255	32	2,578	7	221	1,393
Cyclone II	None	half	8	204	16	1,828	7	184	1,475
	Variable	half	8	204	16	2,013	7	179	1,435
	Erasures	half	8	204	16	3,339	7	164	1,315
	Variable and erasures	half	8	204	16	3,643	8	150	1,201
	None	half	8	255	32	3,237	7	159	1,004

Notes to Table 1–3:

- (1) Stratix II devices use adaptive look-up tables (ALUTs).

Overall resource requirements vary widely depending on the parameter values used. The number of logic elements (LEs) 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 (RS) Compiler using the OpenCore® Plus feature, include these steps in your design flow:

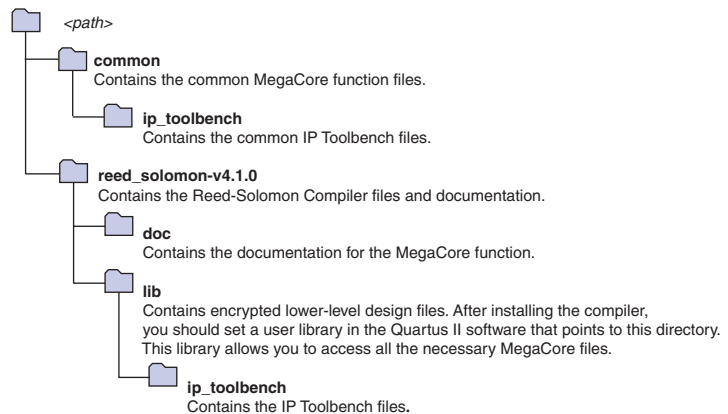
1. Obtain and install the RS Compiler.



For installation instructions, refer to the *Reed-Solomon Compiler v4.1.0 Release Notes*.

Figure 2–1 shows the directory structure after you install the RS Compiler, where *<path>* is the installation directory.

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 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.

Once you have purchased a license for the RS Compiler, the design flow involves the following 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.
4. Perform design verification.

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 involves the following steps:

- “Create a New Quartus II Project” on page 2–3
- “Launch IP Toolbench” on page 2–4
- “Step 1: Parameterize” on page 2–5
- “Step 2: Set Up Simulation” on page 2–8
- “Step 3: Generate” on page 2–10

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. You can also use the Quartus II Web Edition software.
2. Choose **New Project Wizard** (File menu).
3. Click **Next** in the **New Project Wizard Introduction** (the introduction 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\temp\rs_project` directory.
 - b. Specify the name of the project. This walkthrough uses **project** for the project name.



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

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. For Linux and Solaris operating systems, add user libraries by following these steps on the **New Project Wizard: Add Files** page:
 - a. Click **User Library Pathnames**.
 - b. Type `<path>\reed-solomon-4.1.0\lib\` into the **Library name box**, where `<path>` is the directory in which you installed the Reed-Solomon Compiler.
 - c. Click **Add to** add the path to the Quartus II project.
 - d. Click **OK** to save the library path in the project.

7. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.
8. On the **New Project Wizard: Family & Device Settings** page, choose the target device family in the **Family** list.
9. 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:

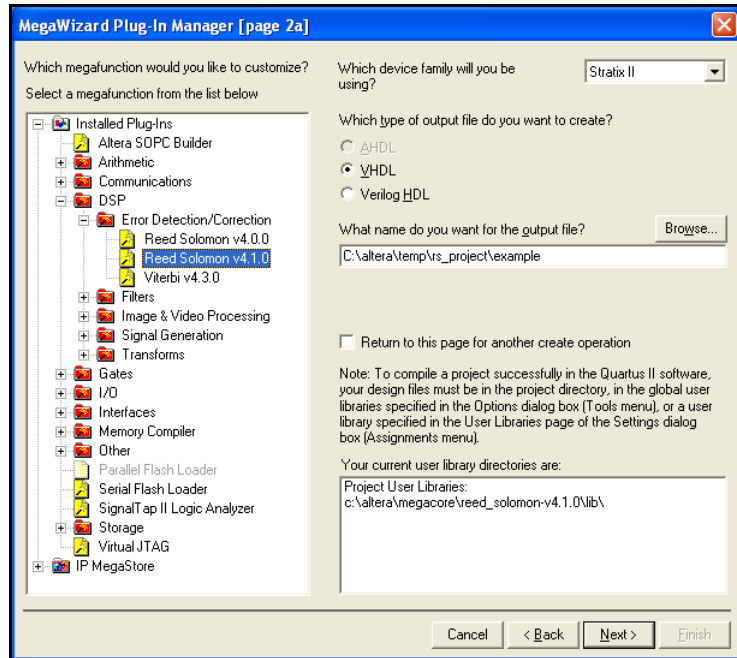
1. Start the MegaWizard® Plug-In Manager by choosing the **MegaWizard Plug-In Manager** command (Tools menu). The **MegaWizard Plug-In Manager** dialog box is displayed.



Refer to the Quartus II Help for more information on how to use the 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 Compiler v4.0.1**.
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 2-2](#) shows the wizard after you have made these settings.

Figure 2–2. Select the Megafunction



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

Step 1: Parameterize

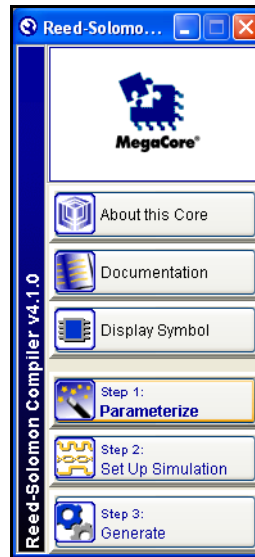
To parameterize your MegaCore function, follow these steps:



For more information on the parameters, refer to [“Parameters”](#) on page 3–9.

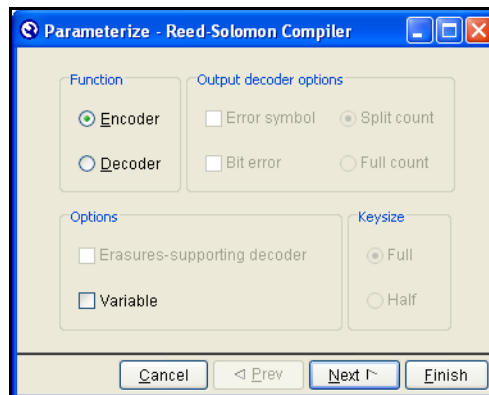
1. Click **Step 1: Parameterize** in IP Toolbench (see [Figure 2-3](#)).

Figure 2-3. IP Toolbench—Parameterize



2. Select **Encoder** or **Decoder** (see [Figure 2-4](#)).

Figure 2-4. Select the Encoder or Decoder



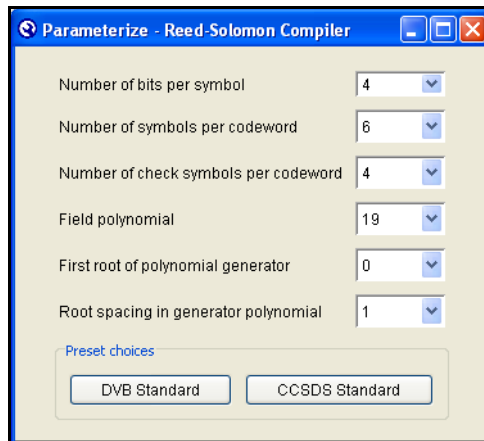
3. For the encoder you can turn on the **Variable** option.



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** and/or **Bit Error** outputs. For the bit error output, select **Split Count** or **Full Count**.
5. Click **Next**.
6. Choose the parameters that define the specific RS codeword that you wish to implement (see [Figure 2–5](#)). 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.

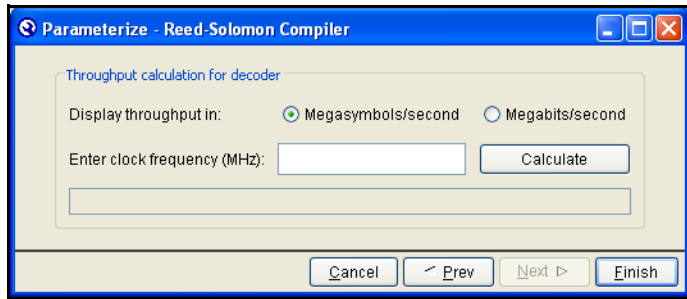
Figure 2–5. Choose the Parameters



7. Click **Next**.

- For a decoder throughput calculation, enter the frequency in MHz, select the desired units, and click **Calculate** (see [Figure 2-6](#)).

Figure 2-6. Throughput Calculator



- 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 file produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

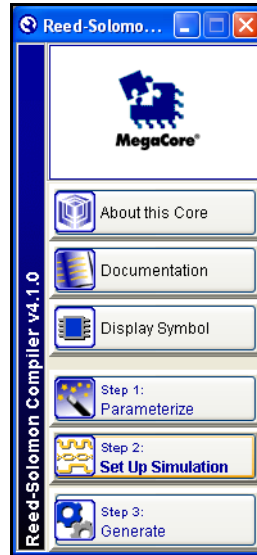


Only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.

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

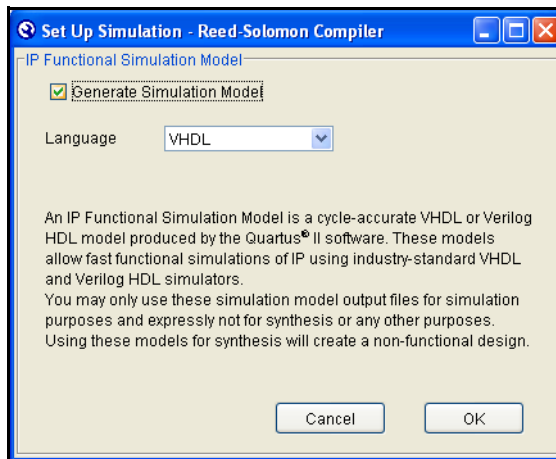
1. Click **Step 2: Set Up Simulation** in IP Toolbench (see [Figure 2-7](#)).

Figure 2-7. Set Up Simulation




2. Turn on **Generate Simulation Model** (see [Figure 2-8](#)).

Figure 2-8. Generate Simulation Model



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

 Choose the same language as your design.

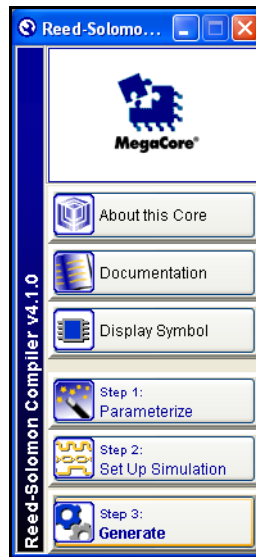
4. Click **OK**.

Step 3: Generate

To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench (see [Figure 2-9](#)).

Figure 2-9. IP Toolbench—Generate



[Figure 2-10](#) shows the generation report.

Figure 2–10. Generation Report

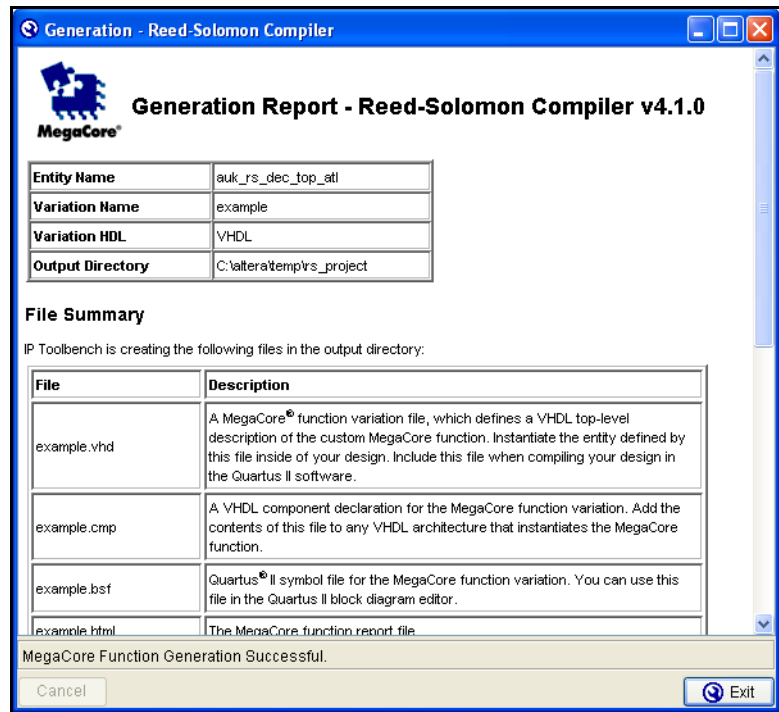


Table 2–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

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>.cmp	A VHDL component declaration file for the MegaCore function variation. Add the contents of this file to any VHDL architecture that instantiates the MegaCore function.
<variation name>.html	MegaCore function report file.
<variation name>.vo or .vho	VHDL or Verilog HDL IP functional simulation model.

Table 2–1. Generated Files (Part 2 of 2) Note (1)	
Filename	Description
<code><variation name>.vhd</code> , or <code>.v</code>	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.
<code><variation name>_bb.v</code>	Verilog HDL black-box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
<code><variation name>_testbench.vhd</code>	The testbench.
<code><variation name>_vsim_script.tcl</code>	Starts the MegaCore function simulation in the ModelSim simulator.
<code>block_period_stim.txt</code>	The testbench stimuli, which change for every block.
<code>rs_encoded_data.txt</code>	Contains the encoded test data.

Notes to Table 2–1:

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

2. After you review the generation report, click **Exit** to close IP Toolbench.

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

Simulate the Design

You can simulate using the IP Toolbench-generated IP functional simulation models. IP Toolbench also generates a customized VHDL or Verilog HDL testbench for your encoder or decoder in your project directory.

To use the IP functional simulation model and the customized testbench with the ModelSim simulator, follow these steps:

1. Start the ModelSim-Altera simulator.
2. Change directory to your Quartus II project directory by typing `cd <directory name>`.
3. To simulate with an IP functional simulation model simulation, type the following command:

```
source <variation name>_vsim_script.tcl
```

The Tcl script performs the following operations:

- Creates a working library **rs_work**
- Compiles your top-level design, the testbench support files, and the testbench with your parameters into **rs_work**
- Executes **vsim** and opens a wave window with the testbench signals

The testbench generates summary report files, **summary_input.txt** (for the decoder only) and **summary_output.txt**, which detail the simulation results.

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 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 3–9, and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Set Up Licensing

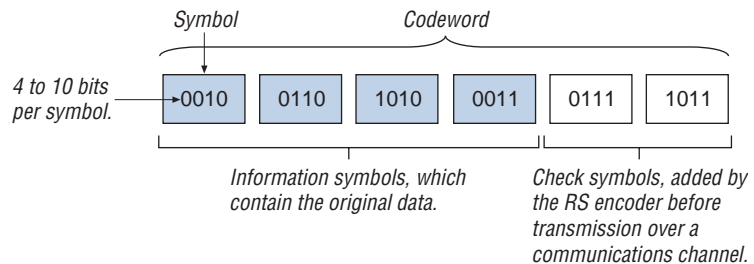
You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance and want to take your design to production.

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.

Functional Description

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 (i.e., 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 (i.e., 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, i.e., $C \leq R$.



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 (`defail` asserted) diminishes (see [Table 3-1 on page 3-7](#)).

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, i.e., 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

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



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

In addition, with 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 3-13 shows the variable option signals.

Interfaces

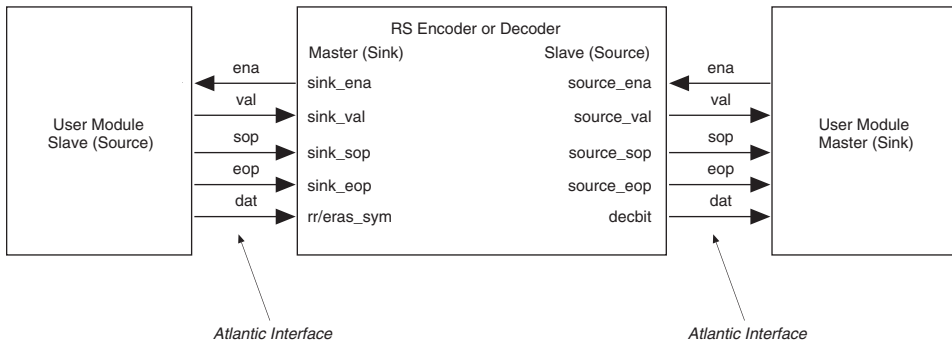
The RS encoder and decoder use the Atlantic™ interface for data input and output. The input is an Atlantic master sink and the output is an Atlantic slave source. The Atlantic interface threshold is set to 1. The Atlantic interfaces allow for flow control.



For more information on the Atlantic interface, refer to the *Atlantic Interface Specification*.

Figure 3-2 shows the RS encoder and decoder Atlantic interfaces.

Figure 3-2. Atlantic Interface



RS Encoder

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

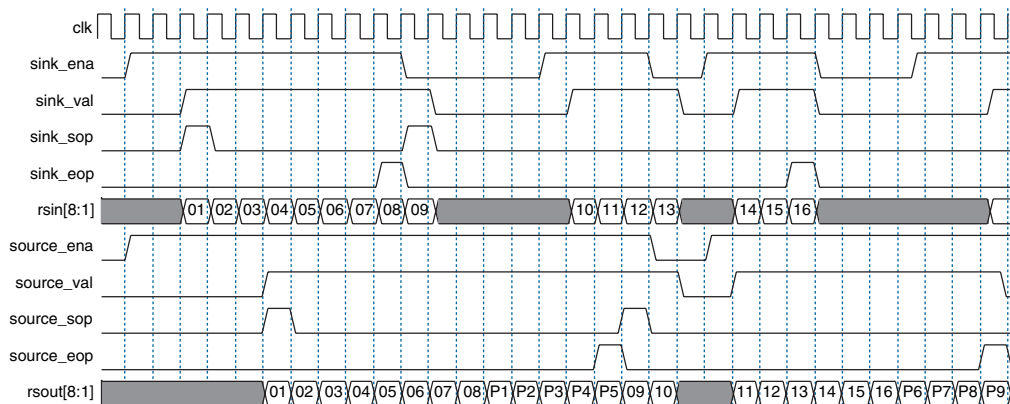


Only assert `sink_val` one clock cycle after the encoder asserts `sink_ena`.

By de-asserting `sink_ena`, the encoder signals that it cannot sink more incoming symbols for a number of symbols after `eop` is signalled at the input. During this time it is generating the check symbols for the current

codeword. Figure 3-3 shows the operation of the RS encoder. The example shows a codeword with eight information symbols and five check symbols.

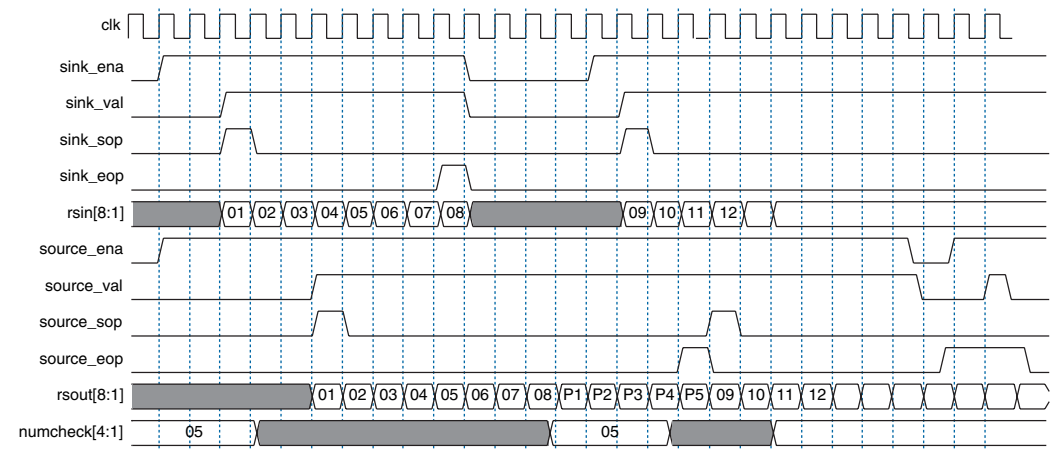
Figure 3-3. Encoder Timing



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 3-4 shows variable encoding.

Figure 3–4. Variable Encoding



RS Decoder

The decoder implements an Atlantic-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 new architecture replaces the previous discrete, streaming, and continuous architectures (see [“Information for Version 3.6.0 Users”](#) on page C–1).

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 .

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:

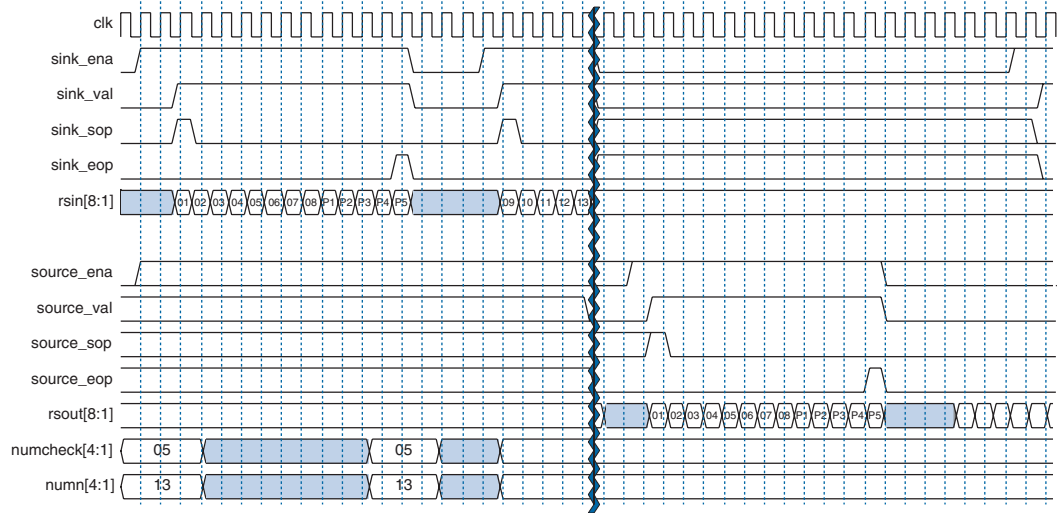
- (1) The decoder may fail to assert `decfail`, for low values of R (4,5, or 6), or when using erasures and the differences between the number of erasures and R is small (4, 5 or 6).

The RS decoder observes Atlantic 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 provides valid data on `rsout` and asserts `source_val`. Also, it indicates the start and end of the codeword with `source_sop` and `source_eop` respectively.

Figure 3–5 shows the operation of the RS decoder.

Figure 3–5. Decoder Timing

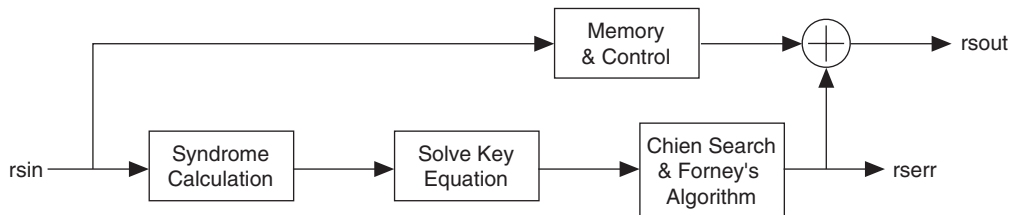


The decoder has the following optional outputs, which you turn on in IP Toolbench:

- Error symbol
- Bit error count

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 3–6 shows the error symbol output.

Figure 3–6. Error Symbol Output



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

- Full count—connects the output `num_err_bit`
- Split count—connects `num_err_bit0` and `num_err_bit1`



For information on these outputs, see [Table 3-8 on page 3-14](#).

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.

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).

Parameter	Value	Description
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 3–6 and Table 3–8 on page 3–14).
Bit error output (1)	On or off.	Specifies the bit error output as either split or full count (see “RS Decoder” on page 3–6 and Table 3–8 on page 3–14).
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.

Parameter	Range	Range (Continuous)	Description
Number of bits per symbol (m)	3 to 12	6 to 10	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 $\min(128, N - 1)$ (1)	4 to $N/7 - 1$	Specifies the number of check symbols per codeword.
Field polynomial	Any valid polynomial (2)		Specifies the primitive polynomial defining the Galois field.
First root of generator polynomial (α_0)	0 to $(2^m - 2)$		Specifies the first root of the generator polynomial.

Table 3–3. RS Codeword Parameters (Part 2 of 2)

Parameter	Range	Range (Continuous)	Description
Root spacing in generator polynomial (<i>a</i>)	Any valid root space (2)		Specifies the space between roots in the generator polynomial.

Notes to Table 3–3:

- (1) Minimum value 4 with half *keysize* and no erasures-supporting option.
- (2) IP 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
<code>clk</code>	<code>clk</code> is the main system clock. The whole MegaCore function operates on the rising edge of <code>clk</code> .
<code>reset</code>	Reset. The entire decoder is asynchronously reset when <code>reset</code> is asserted high. The <code>reset</code> signal is used to reset the entire system. The <code>reset</code> signal must be de-asserted synchronously with respect to the rising edge of <code>clk</code> .

Table 3–5 shows the Atlantic master sink (data input) signals.

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

Table 3-6 shows the Atlantic slave source (data output) signals.

Name	Atlantic Type	Direction	Description
source_ena	ena	Input	Data transfer enable signal. <code>source_ena</code> is driven by the interface master and used to control the flow of data across the interface. <code>ena</code> behaves as a read enable from master to slave. When the slave observes <code>source_ena</code> asserted on the <code>clk</code> rising edge it drives, on the following <code>clk</code> rising edge, the Atlantic data interface signals and asserts <code>source_val</code> . The master captures the data interface signals on the following <code>clk</code> rising edge. If the slave 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 for one clock cycle, 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 “RS Decoder” on page 3-6.

Table 3-7 shows the 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 R (variable option only). The decoder samples <code>numcheck</code> only when <code>sink_sop</code> is asserted.
numn	Variable value of N . Can be any value from the minimum allowable value of N up to the selected value of N (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	
Name	Description
decfail	Indicates non-correctable number of errors. Valid when <code>source_sop</code> is asserted. Atlantic type err.
num_err_sym	Number of symbols errors. Valid when <code>source_sop</code> is asserted.
num_err_bit	Number of bits errors corrected in the codeword. Valid when <code>source_sop</code> is asserted. Connected only when bit error (full count) option is turned on, see “RS Decoder” on page 3–6.
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. The decoder presents these values at the next <code>source_sop</code> assertion (i.e., 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. The decoder presents these values at the next <code>source_sop</code> assertion (i.e., at the next codeword). Connected only when bit error (split count) option is turned on.

MegaCore Verification

The MegaCore verification includes an automated regression test suite, which is described in the following paragraphs.

Tcl scripts drive the simulation at RTL. In a testbench that includes an RS encoder, a channel, and an RS decoder, data is randomly generated and fed to the RS encoder. In the channel, some errors are introduced at various locations of the RS codeword. An account of those errors is kept. The testbench then receives the data decoded by the RS decoder and compares it with the originally transmitted data. The error report and three flag signals are generated: failure to correct; misleading `decfail`; and failure to detect. The first two flags indicate a misbehavior of the MegaCore function—hence a bug. The third signal tracks a condition that happens with a low number of check symbols and when the number of erasures comes close to the number of check symbols.

The test script defines sets of tests that cover a comprehensive set of parameters on RTL VHDL and Verilog HDL simulation. Then synthesis is carried out, and simulation using post-synthesis vital VHDL and Verilog HDL netlist is performed.

Throughput Calculator

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

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

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

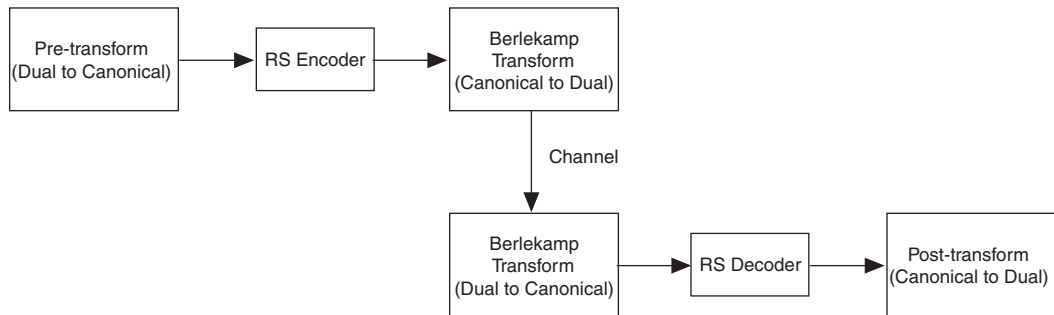
Table 3–9 shows the value of N_C .

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, e.g., when working with the CCSDS standard. To transfer from a canonical-base to a dual-base system, a Berlekamp transform is used, which you will 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.

You must implement a finite state machine (FSM) to control your Atlantic™ slave source interface that connects to the Reed-Solomon (RS) master sink interface.



You need not implement an FSM if you have a fully continuous system and you are not using output flow control.

Altera supplies a reference implementation in VHDL and Verilog HDL in the stimulus and response block files for the encoder and decoder functions (see “[Stimulus & Response Block Files](#)” on page B-4).

Figure B-1 shows the connectivity between your logic and the RS master sink interface. The source control logic is implemented between the stimulus and response block and the RS function at the testbench.

Figure B-1. Atlantic to RS Master Sink Interface

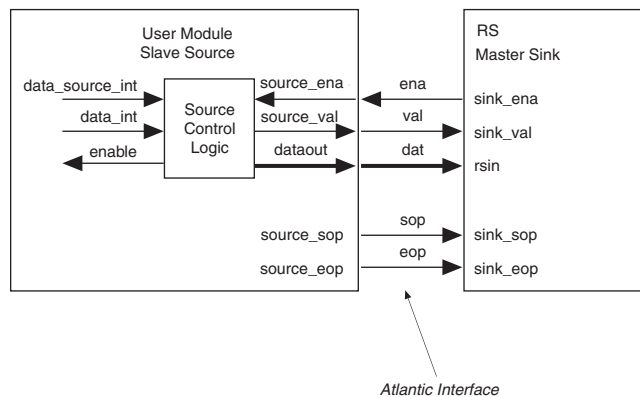
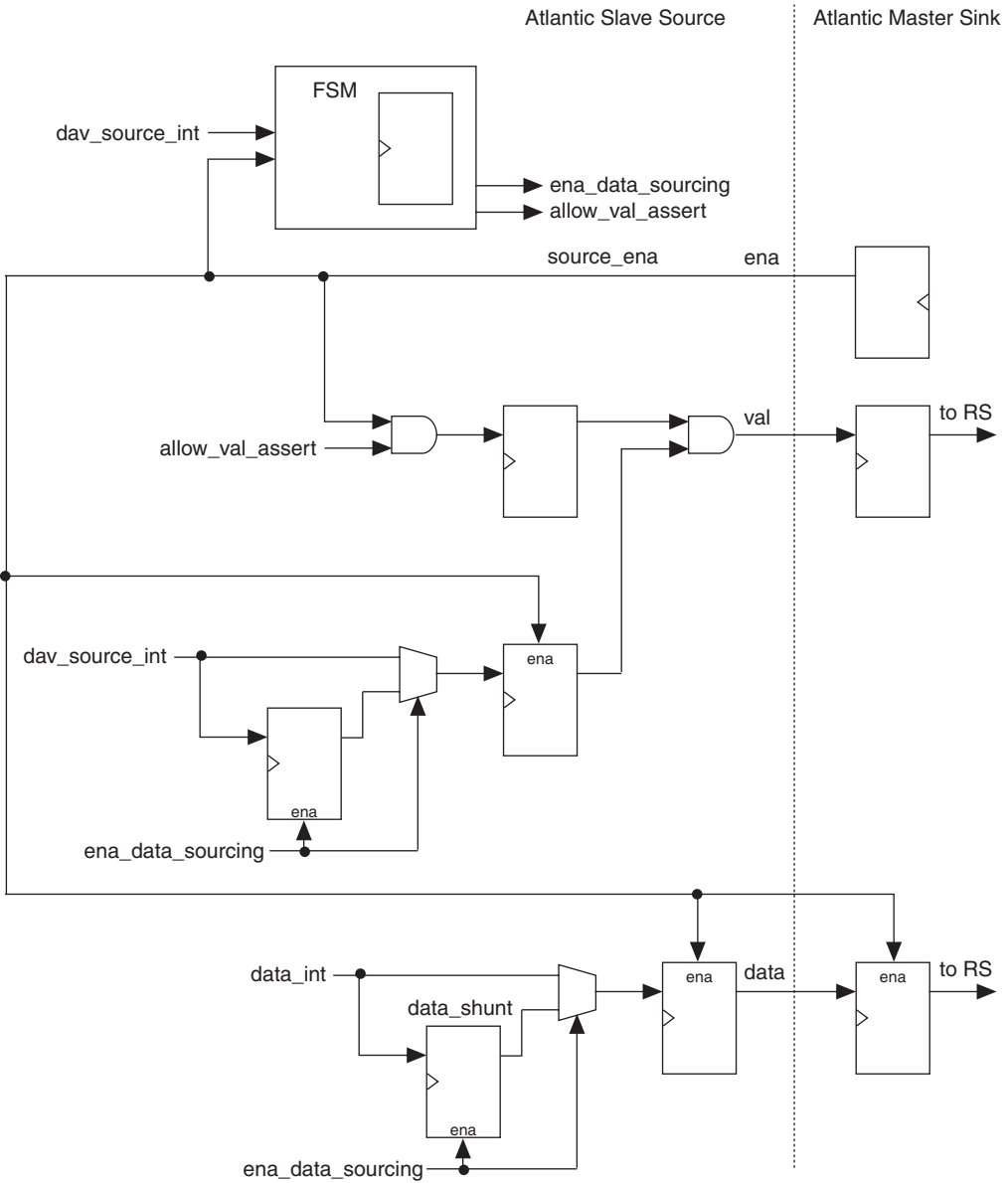


Figure B-2 shows the Atlantic slave source control implementation.

Figure B-2. Atlantic Slave Source Control Implementation



The FSM monitors the `source_ena` and `dav_source_int` signals. The `source_ena` signal comes from the RS's `sink_ena`. The `dav_source_int` signal is an internal signal that indicates that the internal data (`data_int`) is valid. See also [Figure B-4](#) on [page B-4](#).

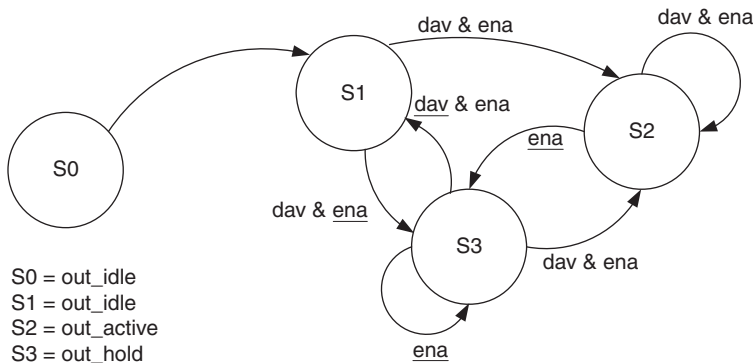
That FSM also controls `ena_data_sourcing` and `allow_val_assert`.

The `ena_data_sourcing` signal enables or disables FSM operation and all the logic prior to `data_int` and `dav_source_int` generation. The `ena_data_sourcing` signal controls all the flip-flops in the `data_int` data path for the shunt buffer and controls your logic prior to `data_int`. When `ena_data_sourcing` is asserted, `data_int` and `dav_source_int` must not change.

The `allow_val_assert` signal controls whether or not `val` can be asserted one clock cycle after the master's `ena` is asserted. If `ena` is asserted, the RS function is ready to receive data. When `sink_ena` remains de-asserted, the function cannot accept any more data.

[Figure B-3](#) shows the state diagram for the FSM.

Figure B-3. State Diagram for the FSM that Controls the Atlantic Slave Source Interface *Note (1)*



S0 = out_idle
 S1 = out_idle
 S2 = out_active
 S3 = out_hold

Monitors:
 dav = dav_source_int
 ena = source_ena

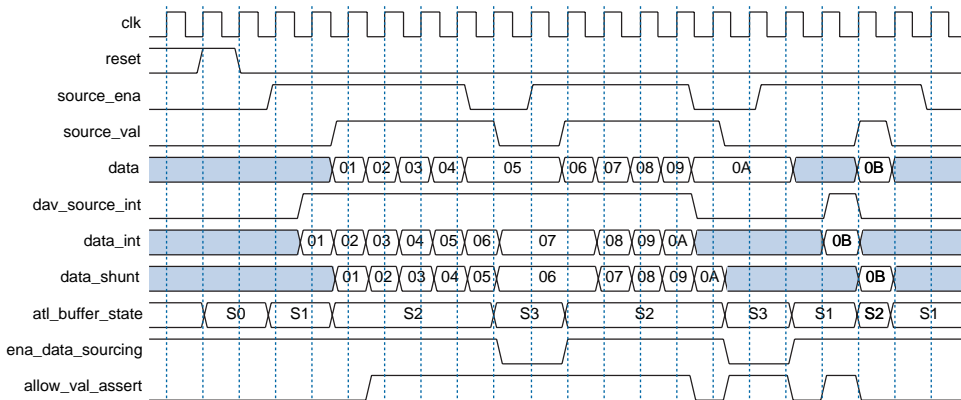
Controls:
 ena_data_sourcing
 allow_val_assert

Note to Figure B-3:

(1) Underlined signal names indicate logic NOT. For example, ena is NOT ena.

Figure B-4 shows the FSM timing diagram.

Figure B-4. FSM and Logic Timing Diagram



Stimulus & Response Block Files

All this logic in VHDL and Verilog HDL is in the following stimulus and response block files:

- Bench_rs_enc_atl_ent.vhd
- Bench_rs_enc_atl_arc_ben.vhd
- Bench_rs_enc_atl.v
- Bench_rs_dec_atl_ent.vhd
- Bench_rs_dec_atl_arc_ben.vhd
- Bench_rs_dec_atl.v

Search the files for the following comments in VHDL:

```
--| START ATLANTIC SOURCE CONTROL LOGIC
--| END ATLANTIC SOURCE CONTROL LOGIC
```

or in Verilog HDL:

```
--// START ATLANTIC SOURCE CONTROL LOGIC
--// END ATLANTIC SOURCE CONTROL LOGIC
```

You should connect the internal data to `data_int`, and provide an internal signal `dav_source_int`. When `dav_source_int` is asserted, the data in `data_int` is valid. Also, you must connect `ena_data_sourcing` to all the enables of the flip-flops of the logic behind `data_int`. When this signal is disabled, the throughput has to stop from the source side because the RS cannot sink data for that time.

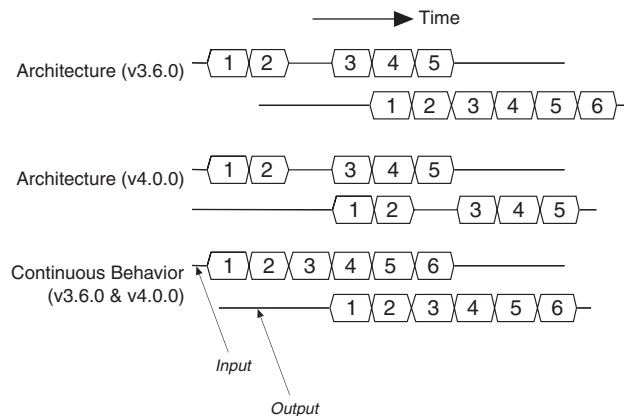
The FSM gets to state *S0* when the asynchronous reset is asserted. After de-asserting reset it jumps to *S1* (*out_idle*). It remains there until the operation starts by asserting *dav_source_int*, which indicates the presence of valid data at *data_int*. The FSM jumps to *S2* (*out_active*) if *source_ena* is asserted (see [Figure B-4](#)). Data is now delivered. After this point there are two options: either internal data is no longer available (*dav_source_int* de-asserted) or the sink side of the RS may signal it cannot sink any more data by de-asserting its *sink_ena* (*source_ena* de-asserted) (see [Figure B-4](#)). The FSM enters *S3* (*out_hold*). It leaves this state when *source_ena* is asserted again. While the FSM is on hold *ena_data_sourcing* remains de-asserted. While *ena_data_sourcing* remains de-asserted, neither *data_int* nor *dav_source_int* change because the logic leading to them is disabled by *ena_data_sourcing*.

The Reed-Solomon (RS) decoder version 3.6.0 had discrete, streaming, and continuous architectures. The new RS decoder version 4.0.0 implements a single Atlantic-based pipelined three-codeword-depth architecture with flow control.


Streaming decoder v3.6.0 users now get an enhanced (faster and smaller) function. Replacing a discrete architecture v3.6.0 with a version 4.0.0 decoder incurs a small penalty in memory usage.

Unlike the streaming and continuous architectures in versions 3.6.0 and earlier, this new decoder does not need new codewords to be fed in for the codewords to be delivered by the decoder (see [Figure C-1](#)).

Figure C-1. Decoder Codewords



The previous streaming architecture interface signals have similar mappings: the `rdyin` signal is like `sink_ena`; `dsin` is like `sink_val`; `outvalid` is like `source_val`; and `dsout` is similar to `source_ena`.

 The timing relationships for the two versions are not the same.

With the new single Atlantic-based pipelined three-codeword-depth architecture with flow control, if `source_ena` remains asserted all the time and the codeword parameter set is of a continuous decoder from version 3.6.0, `sink_ena` never gets de-asserted. This arrangement allows

continuous input of codewords to the v4.0.0 decoder. However, because of the variable latency of the Berlekamp Massey block, there is a possibility of small gaps where `source_val` is not asserted at the output. After a few codewords, the latency eventually reaches the maximum latency of the decoder, when a block that causes the maximum latency has passed through. Afterwards, there are no more gaps or de-assertions in `source_val`. The output timing is therefore data-dependent.

You can create a workaround module, effectively an Atlantic FIFO buffer that sits on the output of the Reed-Solomon decoder so that the combination of decoder and FIFO buffer has a fixed latency throughout and therefore is continuous for output and input.

The length of this FIFO buffer depends on the desired delay and on keysize. To delay the data up to $3 \times N + 4$ from `sink_sop`, to emulate the former continuous decoder, you need a FIFO buffer with depth N . To delay up to the maximum latency of the Berlekamp-Massey block, the depth should be $3 \times R$ for full keysize and $5 \times R$ for half keysize. Where R is the number of check symbols.