Section 1. User Guide

1.1 Introduction

This document presents an implementation of floating-point arithmetic as described in **[1]**. The following floating-point routines for the 56800E device family are implemented (see also **[1]** and **[2]** for detailed description of their functionality):

1. Basic floating-point operations: addition, subtraction, multiplication, division
2. Conversion to and from integer (16-bit and 32-bit) and floating-point format, both round-to-nearest-even and toward-zero versions
3. Comparison functions
4. Rounding functions: `floor`, `ceil`, `round`, `trunc`, `rint`
5. Function for controlling floating-point state as defined in **[2]**: `getround`, `setround`, `testexcept`, `getexceptflag`, `setexceptflag`, `clearexcept`

Floating-point functions are provided in the form of libraries and source code, both C and assembly.

The implementation is prepared for use with the CodeWarrior compiler.

The release contents are divided into a few folders as follows:

- `...\examples` - contains operational examples of use of the software
- `...\lib` - contains floating-point libraries for immediate use
- `...\proj` - contains CodeWarrior project needed for re-build of all libraries
- `...\src` - contains all source files

The implementation demonstrates a good balance between functionality and performance, and for this reason does not strictly follow the floating-point standard described in **[1]**. In particular, the implementation provides a few library variants, each of them differing in compliance level to the standard **[1]**.

The different library variants together with supported floating-point features are described in the table **Table 1-1**

Table 1-1 Floating-Point Library Variants

	Library Variants (library tag is shown)		
Features	**fast**	**balan**	**advan**
Rounding	unspecified/ round to zero	round to nearest even	directed rounding
Non-numerical values	NO†	NO†	YES†
Floating-point state bits	NO	NO	NO
Exception/Traps	NO	NO	NO
Sub-normals	YES	YES	YES
† feature customizable, can be switched on or off depending on defined assembler macros			

Different library variants differ in speed performance. The variant *fast* is the fastest, the variant *balan* is slower, however it exhibits a good balance between speed, accuracy and functionality. The *advan* variant is the slowest one, however offers the highest conformance to the standard.

Due to defined features of different library variants, some functions may have limited functionality.

For example the directed float-float rounding function (`rint`) rounds always toward zero in the fast variant of the library.

Another example - the fast variant does not support rounding mode in a consistent way. For addition, subtraction, multiplication and division the rounding mode may vary from operation to operation resulting in an error of 1 ulp. For other operations (floating and integer conversions) the round-to-zero rounding mode is used (see **1.4.6 Rounding** for more details).

NOTE: *A detailed discussion regarding use of the different floating-point features imposed by the IEEE-754 standard **[1]** is beyond the scope of this document and will not be provided. However, users are reminded that this subject is non-trivial. It is recommended that users familiarize themselves with the appropriate literature in order to use all such features correctly (see **[3]**).*

1.2 Usage

The floating-point libraries should be used by adding a floating-point library to a CodeWarrior project. The CodeWarrior linker will link the project compiled binaries against the added library.

The library files are located in `...\lib` folder. The libraries names are composed as follows:

- `fplib_`*`<library tag>_<memory model>`*

where:

- `fplib_` is a library identifier
- *`<library tag>`* is one of the library tags as shown in **Table 1-1**
- *<memory model>* is memory model as with other CodeWarrior libraries

An example of how to add a floating-point library to a CodeWarrior project is shown in **Table 1-1**. An operational example demonstrating use of the provided floating-point libraries can be found in the `...\examples` folder.

The CodeWarrior linker may report warnings about ambiguous symbols if a floating-point library from the CodeWarrior release is used. If such behaviour is not acceptable the floating-point library from the CodeWarrior release should be removed from the project.

To run correctly, the floating-point libraries require the following:

- Appropriate setting of the OMR register:
  - SA = 0 - saturation mode bit cleared
  - R = 0 - convergent rounding is set
- Inclusion of header file: `fpieee.h` from the`...\src` directory

Other standard headers may require to be included as well (`math.h`, `fenv.h`, `float.h`).

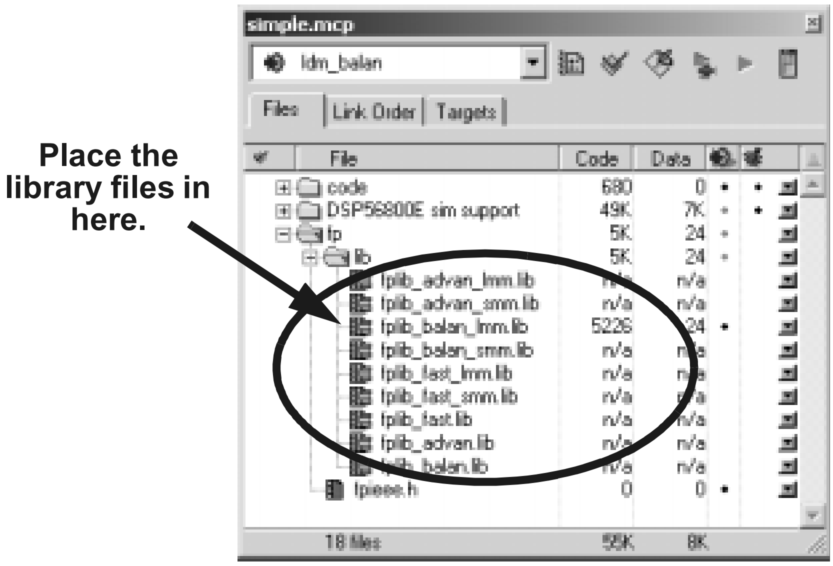

Figure 1-1 Example of Adding Floating-Point Library to Codewarrior Project

The floating-point routines contained in the floating-point libraries can be called in two ways. Firstly, implicitly by the CodeWarrior compiler through ANSI C arithmetic and cast operators. Secondly, explicitly by use of the full names of floating-point functions.

The floating-point function names are composed as follows:

- `__rznv_fp`*`<function tag>`*
- `__rznv_fp`*`<function tag>`*`_`*`<lib. tag><mem. model>`*

where:

- `__rznv_fp` - is a unique identifier
- *`<function tag>`* - is the function tag
- *`<lib. tag>`* - is library tags as shown in **Table 1-1**

- `<mem. model>` - is memory model (`_lmm`, `_smm` or nothing)

The function identifiers are specified in the list below:

- `addf`, `subf`, `mulf`, `divf` - addition, subtraction, multiplication, division
- `ftos`, `ftous`, `ftol`, `ftoul` - conversion of floating-point number to respectively signed short, unsigned short, signed long, unsigned long, toward-zero rounding mode
- `ftosr`, `ftousr`, `ftolr`, `ftoulr` - conversion of floating-point number to respectively signed short, unsigned short, signed long, unsigned long, directed rounding mode
- `stof`, `ustof`, `ltof`, `ultof` - conversion of integer number, respective signed short, unsigned short, signed long, unsigned long to floating-point number
- `gtf`, `gef`, `ltf`, `lef`, `eqf`, `nef` - comparisons, respectively greater, greater equal, lower, lower equal, equal, not equal, the order of arguments is defined as follows: `__rznv_fp<`*`function tag`*`>(x,y) = x `*`op`*` y`, where *op* is an ANSI operator corresponding to a comparison function
- `floorf`, `ceilf`, `roundf`, `truncf`, `rintf` - rounding functions, respectively round down, round up, round to nearest even, round toward 0, directed rounding (according to set rounding mode)
- `getround`, `setround`, `testexcept`, `getexceptflag`, `setexceptflag`, `clearexcept` - function controlling floating-point state (see **[2]**), the standard names (**[2]**) are supported too

It should be noticed that creation of symbol names can be customized as described in **1.3 Advanced Features**.

The library user should pay attention to the following comments about library use.

All functions have been designed to execute as fast as possible in the presence of normalized number as input arguments. In the case where sub-normal numbers are supplied, the execution time may be longer. In any case it should be noted that a frequent appearance of sub-normal numbers in floating-point computation may indicate that an implemented algorithm needs some refinement.

The binaries contained in the provided libraries do not contain symbolic information and are not suitable for debugging. A user wishing to debug the floating-point library functions will have to re-build the libraries with the use of the CodeWarrior project located in the `...\proj` directory.

1.3 Advanced Features

The package provides several advanced features, which can be utilized in order to customize package functionality to specific needs.

All files containing assembly source code of floating-point functions include before any other statements two files: `fpopt_all.asm` and `fpopt_<library tag>.asm`, where `<library tag>` is a library identifier (on of `fast`, `balan`, `advan`). These files must be accessible during compilation and are intended to contain some defines (the `DEFINE` directive) for conditional compilation.

The following defines may be used:

- `CWDFTLIB` - the library tag (`fast`, `balan` or `advan`) of a library variant containing compiler implicit symbols for floating point operations, if `all` is defined, then all library variants will contain the implicit symbols, if `CWDFTLIB` does not contain any of `all`, `fast`, `balan` or `advan`, no library variant will contain implicit compiler symbols. In this case the word `none` is preferred.
- `DFTLIB` - the library tag of a library variant containing the default symbols names (`fast`, `balan` or `advan`), if `all` is defined then all library variants will contain the default symbols, if `DFTLIB` does not equal to one of: `all`, `fast`, `balan` or `advan`, no library variant will contain the default symbols names. In this case the word `none` is preferred.
- `NONNUM` - if defined, will cause for all floating-point functions to handle properly the non-numerical values like infinity and nan, if not defined, non-numerical values will be treated as described in **1.4.2 Non-numerical Values**.

1.4 Supported IEEE-754 Features Description

1.4.1 Format

The implementation uses the single-precision format described in **[1]**. The implementation does not use extended and double precision formats.

1.4.2 Non-numerical Values

Depending on the library variant, the non-numerical values like: NaN (not a number) and Inf (infinity) may be or may not be supported. If supported, the non-numerical values are treated by the floating-point functions as specified in **[1]**.

If the non-numerical values are not supported, they are handled in a special way described below:

If non-numerical values are supplied as input arguments, they are treated as normalized numbers as follows (e is the exponent, f is the mantissa and v is the actual value):

- if $e = 255$ and $f = 0$, then the value is equal to $v = (-1)^s \cdot 2^{128} \cdot (1 \cdot f)$ or $v = (-1)^s \cdot 2^{128} \cdot (1 \cdot 0)$ (Infinity)
- if $e = 255$ and $f \neq 0$, then the value is equal to $v = (-1)^s \cdot 2^{128} \cdot (1 \cdot f)$ (NaN)

Additionally if non-numerical values are not supported, the floating-point functions produce results which are limited by the value corresponding to infinity ($(-1)^s \cdot 2^{128} \cdot (1 \cdot 0)$). In other words, it is not possible to produce a value which is larger in magnitude than a value corresponding to infinity (even if the input arguments would have suggested something oppositely).

This means that there are several operations which are defined as incorrect by **[1]**. Some examples follow (NaN =a NaN number, Inf = Infinity):

- NaN - NaN = 0 (zero)
- NaN + NaN = Inf
- Inf - Inf = 0 (zero)
- Nan*Nan = Inf

If non-numerical values are not supported, the result of division by zero is computed in a special way. In case the denominator is zero, and the numerator is not zero (can be a number, infinity or NaN), the result will be infinity with the sign computed according to provided arguments. In case the denominator is zero and the numerator is zero, the result will be zero with appropriate sign resulting from the division arguments.

1.4.3 Floating-point State

Currently floating-point state is not supported.

1.4.4 Sub-normal Values

The sub-normal values are supported by all library variants.

It is not possible to let the floating-point functions treat the sub-normal values in a different way (for example as zero, so called flushing-to-zero).

1.4.5 Exceptions/Traps

Exception/traps handling is currently not supported. As limited work-around one may use functions handling non-numerical behaviour provided in the file `fpnonnum_56800e.h`.

1.4.6 Rounding

The implementation uses different rounding depending on the floating-point library variant (see **Table 1-1**).

1.4.6.1 The fast variant

All routines provided by the balan and advan variants exhibit consistent rounding modes. The fast variant, in opposite, does not support rounding in a consistent way, which means that depending on arguments and result the actually used rounding mode may vary. Thus the results of computations performed by functions may differ by 1 ulp from a correct value.

For addition, subtraction, multiplication and division the rounding mode is unspecified.

For other functions the round-toward-zero rounding mode is used.

1.4.6.2 The balan variant

All applicable functions follow round-to-nearest-even rounding mode.

For rounding to the nearest even number, the implementation uses the 56800E device hardware function of convergent rounding. It means that the rounding behaviour of the floating-point library function will follow the 56800E device rounding mode bit in the OMR register.

1.4.6.3 The advan variant

The advan variant support various rounding modes (toward zero, toward plus/minus infinity, to nearest even).

The rounding mode can be set by the floating-point state control functions (**[2]**).

With exception of implicit float-to-integer conversions, all functions follow the defined rounding mode.

The implicit float-to-integer conversions follow the toward-zero rounding mode. If round-to-nearest even rounding mode is required, the user is advised to use the appropriate variant of conversion functions (with the suffix `r`: `ftosr`, `ftousr`, `ftolr`, `ftoulr`) by explicit calls.

1.5 Known Issues

The compiler does not generate interrupt wrappers around floating point routines. It may cause unwanted register corruption in interrupt service routines. As work-around, it is necessary to check what registers are used by a particular floating-point routine and make appropriate backup of register on stack. A list of registers used is provided in all assembly source files containing interrupt wrappers with the tag `isr`, for example `fpsrc_56800e_addfisr_balan.asm`.

1.6 Bibliography

1. ANSI/IEEE Std. 754-1985 *IEEE Standard for Binary Floating-Point Arithmetic*
2. ISO/IEC 9899:1999 *Programming languages - C*
3. *What Every Computer Scientist Should Know About Floating-Point Arithmetic* David Goldberg ACM Computing Surveys, Vol 23, No 1, March 1991

Section 2. Floating-Point Function Summary

The floating-point functions summary is provided in a form of a table. The table divides all functions into a few groups. Then for each function, which is identified by its tag (see **1.2 Usage** how to construction the full function name from its tag), types of input arguments and a type of the return value is provided.

2.1 Execution Times

The tables contain the execution time expressed in clock cycles. It is assumed that all floating-point code is located in the internal flash of the device and the clock is set to its maximum value allowed.

Performance figures are provided for three cases, denoting different set of arguments:

- both input arguments are numerical (not de-normalized)
- at least one of the input arguments is de-normalized, but none of them is non-numerical (NaN or infinity)
- at least one of the input argument is non-numerical (NaN or infinity)

For each arguments set, a separate table is created with relevant performance figures.

In case, when a particular library variant is not predicted to work with a specific arguments set, the string N/A is placed in the table instead of a number.

In case, the input argument is an integer type, the performance figures are placed in the table corresponding to the arguments set, when both input arguments are numerical and not de-normalized.

Notes to the tables:

The “?” operator, temporarily used in the tables, has the following meaning:

- if $x = y$, then $x ? y = 0$
- if $x > y$, then $x ? y = 1$

- if $x < y$, then $x ? y = 2$
- if x, y are unordered, then $x ? y = 3$

Table 2-1 Floating-Point Function Summary - both arguments are numerical and not de-normalized

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Basic functions	addf	float, float	float	Floating-point addition	111/111	118/141	136/188
	subf			Floating-point subtraction	118/119	126/149	182/196
	mulf			Floating-point multiplication	101/103	127/130	171/174
	divf			Floating-point division	164/165	186/190	232/259
Comparison Compares two floating-point number by an C operator and returns	cmpf	float, float	short	`cmpf(x,y) = (x ? y)`	46/50	46/48	58/58
	cmpef			`cmpef(x,y) = (x ? y)`	44/48	44/46	57/57
	gtf			`gtf(x,y) = (x > y)`	37/41	36/38	49/49
	gte			`gef(x,y) = (x >= y)`	37/41	36/38	50/50
	ltf			`ltf(x,y) = (x < y)`	37/41	38/40	51/51
	lef			`lef(x,y) = (x <= y)`	38/42	37/39	51/51
	eqf			`eqf(x,y) = (x == y)`	38/42	38/40	50/50
	nef			`nef(x,y) = (x != y)`	37/41	37/39	49/49
Conversion from integer to float	stof	float	signed short	Conversion from an integer type (as shown in argument type) to floating point type	42/42	35/35	44/44
	ustof	float	unsigned short		25/25	20/35	29/44
	ltof	float	signed long		44/44	38/38	48/48
	ultof	float	unsigned long		25/25	21/36	29/44
Conversion from float to integer roune-to-nearest	ftosr	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with directed rounding mode	38/38	38/38	45/45
	ftousr	unsigned short	float		19/19	19/34	26/41
	ftolr	long	float		38/38	38/38	48/48
	ftoulr	unsigned long	float		19/19	20/35	26/41
Conversion from float to integer tround-oward-zero	ftos	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with round-toward-zero rounding mode	36/36	36/36	35/35
	ftous	unsigned short	float		36/36	36/36	37/37
	ftol	long	float		35/35	60/60	72/86
	ftoul	unsigned long	float		33/33	54/54	67/77

Table 2-1 Floating-Point Function Summary - both arguments are numerical and not de-normalized

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Rounding	roundf	float	float	Round to nearest even	26/26	26/26	32/32
	floorf			Round down (rounded number is always less or equal)	25/25	25/25	32/32
	ceilf			Round up (rounded number is always greater or equal)	25/25	25/25	32/32
	truncf			Round toward 0 (rounded number is less or equal in magnitude)	26/26	26/26	33/33
	rint			Directed rounding	32/32	30/30	44/61

Table 2-2 Floating-Point Function Summary - at least one argument is de-normalized and none is non-numerical

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Basic functions	addf	float, float	float	Floating-point addition	110/113	118/143	136/190
	subf			Floating-point subtraction	118/121	126/151	144/198
	mulf			Floating-point multiplication	101/103	127/140	171/187
	divf			Floating-point division	164/171	186/205	232/266
Comparison Compares two floating-point number by an C operator and returns	cmpf	float, float	short	`cmpf(x,y) = (x ? y)`	46/50	46/50	58/62
	cmpef			`cmpef(x,y) = (x ? y)`	44/48	44/48	57/61
	gtf			`gtf(x,y) = (x > y)`	37/41	36/40	49/53
	gte			`gef(x,y) = (x >= y)`	37/41	36/40	50/54
	ltf			`ltf(x,y) = (x < y)`	37/41	38/42	51/55
	lef			`lef(x,y) = (x <= y)`	38/42	37/41	51/55
	eqf			`eqf(x,y) = (x == y)`	38/42	38/42	50/54
	nef			`nef(x,y) = (x != y)`	37/41	37/41	49/53

Table 2-2 Floating-Point Function Summary - at least one argument is de-normalized and none is non-numerical

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Conversion from integer to float	stof	float	signed short	Conversion from an integer type (as shown in argument type) to floating point type	64/64	75/75	114/114
	ustof	float	unsigned short		25/53	20/76	29/115
	ltof	float	signed long		67/67	87/87	127/127
	ultof	float	unsigned long		25/64	21/85	29/123
Conversion from float to integer roune-to-nearest	ftosr	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with directed rounding mode	60/60	60/60	67/67
	ftousr	unsigned short	float		19/47	19/47	26/54
	ftolr	long	float		61/61	61/61	71/71
	ftoulr	unsigned long	float		19/58	20/59	26/65
Conversion from float to integer tround-oward-zero	ftos	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with round-toward-zero rounding mode	N/A	N/A	N/A
	ftous	unsigned short	float		N/A	N/A	N/A
	ftol	long	float		N/A	N/A	N/A
	ftoul	unsigned long	float		N/A	N/A	N/A
Rounding	roundf	float	float	Round to nearest even	86/86	86/86	92/92
	floorf			Round down (rounded number is always less or equal)	100/101	100/101	107/108
	ceilf			Round up (rounded number is always greater or equal)	100/101	100/101	107/108
	truncf			Round toward 0 (rounded number is less or equal in magnitude)	55/55	55/55	62/62
	rint			Directed rounding	61/61	90/90	90/128

Table 2-3 Floating-Point Function Summary - at least one argument is non-numerical

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Basic functions	addf	float, float	float	Floating-point addition	89/113	N/A	N/A
	subf			Floating-point subtraction	97/121	N/A	N/A
	mulf			Floating-point multiplication	103/103	N/A	N/A
	divf			Floating-point division	164/171	N/A	N/A
Comparison Compares two floating-point number by an C operator and returns	cmpf	float, float	short	`cmpf(x,y) = (x ? y)`	42/50	N/A	N/A
	cmpef			`cmpef(x,y) = (x ? y)`	40/48	N/A	N/A
	gtf			`gtf(x,y) = (x > y)`	37/41	N/A	N/A
	gte			`gef(x,y) = (x >= y)`	37/41	N/A	N/A
	ltf			`ltf(x,y) = (x < y)`	37/41	N/A	N/A
	lef			`lef(x,y) = (x <= y)`	38/42	N/A	N/A
	eqf			`eqf(x,y) = (x == y)`	38/42	N/A	N/A
	nef			`nef(x,y) = (x != y)`	37/41	N/A	N/A
Conversion from integer to float	stof	float	signed short	Conversion from an integer type (as shown in argument type) to floating point type	42/42	N/A	N/A
	ustof	float	unsigned short		25/40	N/A	N/A
	ltof	float	signed long		44/44	N/A	N/A
	ultof	float	unsigned long		25/40	N/A	N/A
Conversion from float to integer roune-to-nearest	ftosr	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with directed rounding mode	38/38	N/A	N/A
	ftousr	unsigned short	float		19/34	N/A	N/A
	ftolr	long	float		38/38	N/A	N/A
	ftoulr	unsigned long	float		19/34	N/A	N/A
Conversion from float to integer tround-oward-zero	ftos	signed short	float	Conversion from the floating-point type to an integer type (as shown in argument type) with round-toward-zero rounding mode	N/A	N/A	N/A
	ftous	unsigned short	float		N/A	N/A	N/A
	ftol	long	float		N/A	N/A	N/A
	ftoul	unsigned long	float		N/A	N/A	N/A

Table 2-3 Floating-Point Function Summary - at least one argument is non-numerical

Function Group	Function Tags	Arguments	Return	Description	Execution Time MIN/MAX [clock cycles]		
					fast	balan	advan
Rounding	roundf	float	float	Round to nearest even	26/26	N/A	N/A
	floorf			Round down (rounded number is always less or equal)	25/25	N/A	N/A
	ceilf			Round up (rounded number is always greater or equal)	25/25	N/A	N/A
	truncf			Round toward 0 (rounded number is less or equal in magnitude)	26/26	N/A	N/A
	rint			Directed rounding	32/32	N/A	N/A

IMPORTANT. Read the following Freescale Software License Agreement ("Agreement") completely. By using the product you indicate that you accept the terms of this Agreement.

FREESCALE SOFTWARE LICENSE AGREEMENT

This is a legal agreement between you (either as an individual or as an authorized representative of your employer) and Freescale Semiconductor, Inc. (“Freescale”). It concerns your rights to use this file and any accompanying written materials (the “Software”). In consideration for Freescale allowing you to access the Software, you are agreeing to be bound by the terms of this Agreement. If you do not agree to all of the terms of this Agreement, do not download the Software. If you change your mind later, stop using the Software and delete all copies of the Software in your possession or control. Any copies of the Software that you have already distributed, where permitted, and do not destroy will continue to be governed by this Agreement. Your prior use will also continue to be governed by this Agreement.

LICENSE GRANT. The Software may contain two types of programs: (i) programs enabling you to design a system (“System Designs”), and (ii) programs that could be executed on your designed system (“System Software”). Your rights in these distinct programs differ. With respect to System Designs, Freescale grants to you, free of charge, the non-exclusive, non-transferable right to use, reproduce, and prepare derivative works of the System Designs for the sole purpose of designing systems that contain a programmable processing unit obtained directly or indirectly from Freescale (“Freescale System”). You may not distribute or sublicense the System Designs to others; however, you may sell Freescale Systems designed using the System Design. Freescale does not grant to you any rights under its patents to make, use, sell, offer to sell, or import systems designed using the System Designs. That is beyond the scope of this Agreement. With respect to System Software, Freescale grants to you, free of charge, the non-exclusive, non-transferable right use, reproduce, prepare derivative works of the System Software, distribute the System Software and derivative works thereof in object (machine-readable) form only, and to sublicense to others the right to use the distributed System Software exclusively with Freescale Systems. You must prohibit your sublicensees from translating, reverse engineering, decompiling, or disassembling the System Software except to the extent applicable law specifically prohibits such restriction. If you violate any of the terms or restrictions of this Agreement, Freescale may immediately terminate this Agreement, and require that you stop using and delete all copies of the Software in your possession or control. You are solely responsible for systems you design using the Software.

COPYRIGHT. The Software is licensed to you, not sold. Freescale owns the Software, and United States copyright laws and international treaty provisions protect the Software. Therefore, you must treat the Software like any other copyrighted material (e.g., a book or musical recording). You may not use or copy the Software for any other purpose than what is described in this Agreement. Except as expressly provided herein, Freescale does not grant to you any express or implied rights under any Freescale or third party patents, copyrights, trademarks, or trade secrets. Additionally, you must reproduce and apply any copyright or other proprietary rights notices included on or embedded in the Software to any copies or derivative works made thereof, in whole or in part, if any.

SUPPORT. Freescale is NOT obligated to provide any support, upgrades or new releases of the Software. If you wish, you may contact Freescale and report problems and provide suggestions regarding the Software. Freescale has no obligation whatsoever to respond in any way to such a problem report or suggestion. Freescale may make changes to the Software at any time, without any obligation to notify or provide updated versions of the Software to you.

NO WARRANTY. TO THE MAXIMUM EXTENT PERMITTED BY LAW, FREESCALE EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE. THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. YOU ASSUME THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE SOFTWARE, OR ANY SYSTEMS YOU DESIGN USING THE SOFTWARE (IF ANY). NOTHING IN THIS AGREEMENT MAY BE CONSTRUED AS A WARRANTY OR REPRESENTATION BY FREESCALE THAT THE SOFTWARE OR ANY DERIVATIVE WORK DEVELOPED WITH OR INCORPORATING THE SOFTWARE WILL BE FREE FROM INFRINGEMENT OF THE INTELLECTUAL PROPERTY RIGHTS OF THIRD PARTIES.