Advance Information

MPC180SWUG/D
Rev. 0.1, 2/2003

MPC180 Security Co-Processor Software Reference Guide

This document describes the interface components for accessing the cryptographic hardware accelerators provided by the MPC180 integrated circuit. It covers the following topics:

Topic	Page
Section 1, "General Overview"	1
Section 2, "MPC180Init Module"	2
Section 3, "MPC180Pkha Module"	3
Section 4, "MPC180Des Module"	8
Section 5, "MPC180Mdha Module"	20
Section 6, "MPC180Afha Module"	24
Section 7, "MPC180Rng Module"	26
Section 8, "MPC180 Interface Module Porting"	27
Section 9, "IPsec and IKE"	30
Section 10, "vxWorks"	30

1 General Overview

The MPC180Lib Interface Module provides a 'C' language programming interface for accessing the cryptographic hardware accelerators provided by the MPC180. Table 1-1 shows the six modules that make up the MPC180Lib Interface.

Table 1-1. MPC180 Programming Interface

	Module	Function
MPC180Lib	MPC180Init	Initialization and Interrupt Handling code
	MPC180Pkha	Public Key Encryption—ECC and RSA
	MPC180Des	DES Encryption
	MPC180Mdha	Hash and HMAC
	MPC180Afha	RC4 Compatible Encryption
	MPC180Rng	Random Number Generation.

The MPC180 is designed to work with the vxWorks 5.4 operating system. Detail on porting may be found in Section 8, “MPC180 Interface Module Porting.”

2 MPC180Init Module

As shown in Table 2-1, the MPC180Init module contains the initialization routines and the interrupt service routine (ISR). The initialization routines are broken into two, one to be used before the operating system is running (mpc180InitHw) and one to be used after the operating system is running (mpc180InitHw2). The routines are described in the sections that follow.

Table 2-1. MPC180Init Module

Module	Routine
MPC180Init	mpc180InitHw
	mpc180InitHw2
	mpc180Initisr

2.1 MPC180Init Routines

The following sections describe the functions that are used to initialize the MPC180 hardware.

2.1.1 mpc180InitHw()

This function is used to initialize the MPC180 hardware before the operating system kernel is running. It resets the MPC180 and masks all interrupts.

```
void mpc180InitHw
(
)
```

Returns: N/A

2.1.2 mpc180InitHw2()

This function is used to initialize the MPC180 hardware and interface modules after the operating system kernel is running. It creates the necessary semaphores and connects the interrupt routine.

```
void mpc180InitHw2
(
)
```

Returns: N/A

2.1.3 mpc180Int()

This function is the interrupt service routine that services the single interrupt line for the MPC180. It determines the type of interrupt that has occurred and gives the appropriate semaphore.

```
void mpc180Int
(
)
```

Returns: N/A

3 MPC180Pkha Module

The MPC180Pkha module performs many advanced mathematical functions to support both RSA and ECC public key cryptographic algorithms. ECC is supported in both F_2m (polynomial-basis) and F_p.

As shown in Table 3-1, the PKHA software module is made up of three sub modules.

Table 3-1. MPC180PkhaUtil Module

Module	Sub Module	Routine
MPC180Pkha	mpc180PkhaUtil	mpc180PkhaLoadModSize
		mpc180PkhaReadModSize
		mpc180PkhaLoadSubReg
		mpc180PkhaReadSubReg
		mpc180PkhaCopySubReg
		mpc180PkhaLoadReg
		mpc180PkhaReadReg
		mpc180PkhaCopyReg
		mpc180PkhaClearMemory
		mpc180PkhaR^2
		mpc180PkhaR$_p$R$_N$
	mpc180PkhaEcc	mpc180EccMultkPtoQ
		mpc180EccAddPto
		mpc180EccDoubleQ
		mpc180EccModularAdd
		mpc180EccModularSubtract
		mpc180EccModularMulitply
		mpc180EccModularMultiply2
	mpc180PkhaRsa	mpc180RsaExpA
		mpc180RsaModularAdd
		mpc180RsaModularSubtract
		mpc180RsaModularMultiply
		mpc180RsaModularMultiply2

The first of these sub modules, mpc180PkhaUtil, allows users to load and read the large number registers which are present on the chip, set/read the modulus size, clear the register memory, and perform supporting calculations. The sub register functions are able to load/read up to 512 bits (or 32 16bit digits) from the sub registers A0, A1, A2, A3, B0, B1, B2, B3, N0, N1, N2, and N3. The full register functions are able to load/read up to 2048 bits (or 128 16bit digits) from the registers A, B, and N. In addition, two supporting calculations may be performed for finding R^2 mod N and R_pR_N mod P.

The second of these sub modules, MPC180Ecc, is used to perform calculations for ECC, or Elliptic Curve Cryptography. Two levels of calculation may be performed.

- The first is high level and performs elliptic curve point multiplication. These point multiplies may be performed in either F_p or F_2m basis and using either projective or affine coordinates. The second level of operations allows the application to perform elliptic curve point additions and point doubles. These operations require that the application work in projective coordinates, and all inputs and outputs of these operations are expected to be in the Montgomery residue system. These operations are used internally by the point multiplication, and are provided to the user, if they are needed.

- The second level of operations allows the application to perform modular multiplication, addition, and subtraction in either F_p or F_2m basis. Notice that the module does not provide the Point Add or Point Double routines that are used internally by the MPC180 to perform Point Multiplication, but which have extremely limited value to an application program.

The third, and final, of these sub modules, MPC180Rsa, is used to perform calculations for modular arithmetic (such as RSA, DSA, and Diffie-Hellman). Two levels of calculation may be performed:

- The first level is high level and performs modular exponentiation, or A ^{EXP} mod N.
- The second level allows the application to perform modular multiplication, addition, and subtraction. This is implemented with the functions mpc180RsaModularMultiply(), mpc180RsaModularMultiply2(), mpc180RsaModularAdd (), and mpc180RsaModularSubtract (). Normally, these use the full register set (512 bits). However, the calls do support sub-register operations. These are provided to support Chinese Remainder Theorem (CRT) operations, which often generate intermediate values that must be stored for later use. By using sub registers, the need to store and retrieve these values from MPC180 memory is alleviated.

For all of the PKHA routines, the **modSize** register specifies the maximum size of the modulus in 16 bit digits. It may be set by calling the mpc180PkhaLoadModSize(). If the number of bits in the modulus is not evenly divisible by 16, those remaining bits above the evenly-divisible number of bits constitutes an entire 16 bit block in so far as setting **modSize** is concerned. Each routine has a minimum number of digits it may work with, and the maximum is 32 for ECC operations and 128 for RSA operations.

The following is an example of the mpc180Pkha module performing an exponentiation using the mpc180Util and mpc180Rsa routines:

```
PKHA_NUM number, exponent, modulus, result;
int modulus_size;
int exponent_size;
     ∑
     ∑
     ∑
mpc180PkhaLoadModSize(modulus_size);
mpc180PkhaLoadReg(modulus, PKHA_REGN);
mpc180PkhaLoadReg(number, PKHA_REGA);
mpc180RsaExpA(exponent, exponent_size, 0, 0, 0);
mpc180PkhaReadReg(result, PKHA_REGB);
```

3.1 MPC180Pkha Routines

The following sections describe the functions that are used to set and read the size of the modulus, load and read PKHA sub registers, and copy from the PKHA sub registers to another register.

3.1.1 mpc180PkhaLoadModSize()

This function is used to set the size of the modulus used for all calculations. It is specified as the number of 16 bit digits in the modulus. If the number of bits in the modulus is not evenly divisible by 16, those remaining bits above the evenly-divisible number of bits constitutes an entire 16-bit block in so far as setting **modSize** is concerned. Each routine has minimum number of digits it may work with, and the maximum is 32 for ECC operations and 128 for RSA operations.

```
STATUS mpc180PkhaLoadModSize
(
unsigned int modSize
)
```

Returns: OK, or ERROR if **modSize** > 128.

3.1.2 mpc180PkhaReadModSize()

This function is used to read the size of the modulus (set by mpc180PkhaLoadModSize()) used for all calculations.

```
unsigned int mpc180PkhaReadModSize
(
)
```

Returns: Current value of **modSize**.

3.1.3 mpc180PkhaLoadSubReg()

This function is used to load one of the MPC180 PKHA sub registers for a calculation. The value to load is specified in x, with the current modSize parameter specifying the number of 16 bit digits in the value x, and regSel specifies into which of the sub registers to load the number (PKHA_B0 = 0, PKHA_B1 = 1, PKHA_B2 = 2, PKHA_B3 = 3, PKHA_A0 = 4, PKHA_A1 = 5, PKHA_A2 = 6, PKHA_A3 = 7, PKHA_N0 = 8, PKHA_N1 = 9, PKHA_N2 = 10, PKHA_N3 = 11).

```
void mpc180PkhaLoadSubReg
(
PKHA_SUBNUM x,
unsigned int regSel
)
```

Returns one of the following:

- OK
- ERROR if **modSize** > 32

3.1.4 mpc180PkhaReadSubReg()

This function is used to read one of the MPC180 PKHA sub registers to obtain a result of a calculation. The value read is returned in x, with the current **modSize** parameter specifying the number of 16-bit digits in the value x, and **regSel** specifies from which of the sub registers to read the number (PKHA_B0 = 0, PKHA_B1 = 1, PKHA_B2 = 2, PKHA_B3 = 3, PKHA_A0 = 4, PKHA_A1 = 5, PKHA_A2 = 6, PKHA_A3 = 7, PKHA_N0 = 8, PKHA_N1 = 9, PKHA_N2 = 10, PKHA_N3 = 11).

```
unsigned int mpc180PkhaReadSubReg
(
PKHA_SUBNUM x,
unsigned int regSel
)
```

Returns one of the following:

- **modSize** parameter specifying the number of 16-bit digits in the value x
- ERROR if **modSize** > 32

3.1.5 mpc180PkhaCopySubReg()

This function is used to copy one of the MPC180 PKHA sub registers to another. The current **modSize** parameter specifies the number of 16 bit digits to copy. **regSrcSel** specifies from which of the sub registers to copy (PKHA_B0 = 0, PKHA_B1 = 1, PKHA_B2 = 2, PKHA_B3 = 3, PKHA_A0 = 4, PKHA_A1 = 5, PKHA_A2 = 6, PKHA_A3 = 7, PKHA_N0 = 8, PKHA_N1 = 9, PKHA_N2 = 10, PKHA_N3 = 11), and **regDestSel** specifies the destination sub register.

```
STATUS mpc180PkhaCopySubReg
    (
    unsigned int regSrcSel,
    unsigned int regDestSel
    )
```

Returns one of the following:

- OK
- ERROR if **modSize** > 32

3.1.6 mpc180PkhaLoadReg()

This function is used to load one of the MPC180 PKHA full registers for a calculation. The value to load is specified in x, with the current **modSize** parameter specifying the number of 16-bit digits in the value x. **regSel** specifies into which of the full registers to load the number (PKHA_B = 0, PKHA_A = 1, PKHA_N = 2).

```
STATUS mpc180PkhaLoadReg
(
PKHA_NUM x,
unsigned int regSel
)
```

Returns: OK

3.1.7 mpc180PkhaReadReg()

This function is used to read one of the MPC180 PKHA full registers to obtain a result of a calculation. The value read is returned in x, with the current **modSize** parameter specifying the number of 16 bit digits in the value x. **regSel** specifies into which of the full registers to load the number (PKHA_B = 0, PKHA_A = 1, PKHA_N = 2).

```
unsigned int mpc180PkhaReadReg
(
PKHA_NUM x,
unsigned int regSel
)
```

Returns: **modSize** parameter specifying the number of 16-bit digits in the value x.

3.1.8 mpc180PkhaCopyReg()

This function is used to copy one of the MPC180 PKHA full registers to another. The current **modSize** parameter specifies the number of 16 bit digits to copy. **regSrcSel** specifies from which of the full registers to copy (PKHA_B = 0, PKHA_A = 1, PKHA_N = 2), and **regDestSel** specifies the destination full register.

```
void mpc180PkhaCopyReg
    (
    unsigned int regSrcSel,
    unsigned int regDestSel
    )
```

Returns: N/A

3.1.9 mpc180PkhaClearMemory()

This routine clears all of the RAM memory locations in the PKHA. This includes the A, B, and N RAMs. All RAM locations are set to zero. This routine is automatically invoked following a reset.

```
STATUS mpc180PkhaClearMemory
(
)
```

Returns: OK, or ERROR if no response from the MPC180.

3.1.10 mpc180PkhaR2()

This function is used to calculate R^2 mod N, where $R = 2^{16D}$ and D is the number of 16 bit digits in the modulus vector N.

The input parameters are expected to be loaded, using the mpc180LoadNReg() function, in the following registers.

Parameter	Register
N (modulus)	N0-3

NOTE

For this routine to execute, the upper 16 bits of N must not be 0.

The output parameters are left in the following registers and may be read with the function mpc180ReadBReg().

Parameter	Register
R^2 mod N	B0-3

This function works with a minimum of 4 digits, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 4 and 128.

```
STATUS mpc180PkhaR2
(
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **modSize** < 4
  - the upper 16 bits of the modulo are 0
  - no response from the MPC180

3.1.11 mpc180PkhaRpRn()

This function is used to calculate R_pR_N mod P, where $R_p = 2^{16D}$ and $R_N = 2^{16E}$, D is the number of 16-bit digits in the modulus p, E is the number of 16-bit digits in the modulus N, and D + 4 < E.

The input parameters are expected to be loaded, using the mpc180LoadNReg() function, in the following registers.

Parameter	Register
P (prime modulus)	N0-3

The output parameters are left in the following registers and may be read with the function mpc180ReadBReg().

Parameter	Register
R_pR_N mod P	B0-3

pModSize and **nModSize** specify the maximum size of the modulus in 16-bit digits. The calculation works with a minimum of 5 digits, so the value of **pModSize** and **nModSize** should be between 5 and 128. This function does not depend on mpc180PkhaLoadModSize to load the **modSize** register, but uses the input parameters instead. The **modSize** register is left set to the **pModSize** specified after execution.

```
STATUS mpc180PkhaRpRn
(
unsigned int pModSize,
unsigned int nModSize
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — pModSize < 4
  - — pModSize > 128
  - — nModSize < 4
  - — nModSize > 128
  - — (pModSize + 4 >= nModSize)
  - — No response from the MPC180

3.2 MPC180PkhaEcc Routines

The following sections describe the functions that perform the elliptic curve point multiplication and addition, ECC modular addition and subtraction, and the Montgomery Modular Multiplication.

3.2.1 mpc180EccMultkPtoQ()

This function performs an elliptic curve point multiply of Q = k x P, where Q =(X_2,Y_2,Z_2) and P = (X_1,Y_1,Z_1).

The parameter **basis** specifies if the point multiply is to be done in the F_p basis (basis = PKHA_F_p) or the F_2m basis (basis = PKHA_F_2m).

The parameter **coordinate** specifies if the point multiply is done with input parameters in projective (coordinate = PKHA_PROJ) or affine (coordinate = PKHA_AFF) coordinates.

The input parameters are expected to be loaded, using the mpc180LoadSubReg function, in the registers specified in the table below.

Parameter	Register
X_1	A0
Y_1	A1
Z_1	A2 (Set to 1 automatically if coordinate=PKHA_AFF)
a elliptic curve parameter	A3
b elliptic curve parameter	B0
R^2 mod N	B1
modulus or prime P	N0

The output parameters are left in the following registers and may be read with the function mpc180ReadSubReg().

Parameter	Register
X_2	B1
Y_2	B2
Z_2	B3
${Z_2}^2$	A2 (when coordinate = PKHA_AFF)
${Z_2}^3$	A3 (when coordinate = PKHA_PROJ)

All of the input numbers are expected to be in standard format, not Montgomery residue system.

When **coordinate** = PKHA_AFF, output consists of the projective coordinate values X_2,Y_2, Z_2 in Montgomery residue system. To put these coordinates in their affine form, the following equations should be used along with the returned values for ${Z_2}^2$ and ${Z_2}^3$:

$$x = X_2 / {Z_2}^2$$

$$y = Y_2 / {Z_2}^3$$

Since the MPC180 does not support the inverse function, it is the responsibility of the host processor to find $(Z^2)^{-1}$ and $(Z^3)^{-1}$ by using any available module-n inversion techniques. Once this is accomplished, the programmer may then use these values and the function mpc180EccFpModularMultiply2() to calculate the required values.

The input parameter **k** is the scalar multiplier. The point multiply works with a minimum modulus size of 5, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) between 5 and 32. **ksize** specifies the size of the multiplier in 32-bit words. The parameter **kLswToMsw** is a boolean which specifies the ordering of words in the input parameter **k**.If the order of words is “LSW is lowest in memory,” **kLswToMsw** should be set to TRUE; if the order is “MSW is lowest in memory,” it should be set to FALSE.

```
STATUS mpc180EccMultkPtoQ
(
PKHA_SUBNUM k,
unsigned int kSize,
unsigned int basis,
unsigned int coordinate,
BOOLEAN kLswToMsw
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **k** < 5
  - **modSize** < 5
  - **modSize** > 32
  - no response from the MPC180

3.2.2 mpc180EccAddPtoQ()

This function performs an elliptic curve point addition of R = P x Q, where Q = (X_2,Y_2,Z_2), P = (X_1,Y_1,Z_1), and R = (X_3,Y_3,Z_3)

The parameter **basis** specifies if the point multiply is to be done in the F_p basis (basis = PKHA_F_p) or the F_2m basis (basis = PKHA_F_2m).

The input parameters are expected to be loaded, using the mpc180LoadSubReg() function, in the following registers.

Parameter	Register
X'_1	A0
Y'_1	A1
Z'_1	A2
a elliptic curve parameter	A3
b elliptic curve parameter	B0
X'_2	B1
Y'_2	B2
Z'_2	B3
modulus or prime P	N0

The output parameters are left in the following registers and may be read with the function mpc180ReadSubReg().

Parameter	Register
X'_3	B1
Y'_3	B2
Z'_3	B3
X'_1	A0

Parameter	Register
Y'_1	A1
Z'_1	A2
a elliptic curve parameter	A3
b elliptic curve parameter	B0

All of the input/output numbers are expected to be in Montgomery residue system, not standard format.

The point multiply works with a minimum modulus size of 5, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 32.

```
STATUS mpc180EccAddPtoQ
(
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **k** < 5
  - **modSize** < 5
  - **modSize** > 32
  - no response from the MPC180

3.2.3 mpc180EccDoubleQ()

This function performs an elliptic curve point addition of R = 2 x Q, where Q = (X_1,Y_1,Z_1) and R = (X_3,Y_3,Z_3)

The parameter **basis** specifies if the point multiply is to be done in the F_p basis (basis = PKHA_F_p), or the F_2m basis (basis = PKHA_F_2m).

The input parameters are expected to be loaded, using the mpc180LoadSubReg function, in the following registers.

Parameter	Register
a elliptic curve parameter	A3
b elliptic curve parameter	B0
X'_1	B1
Y'_1	B2
Z'_1	B3
modulus or prime P	N0

The output parameters are left in the following registers and may be read with the function mpc180ReadSubReg().

Parameter	Register
X'_3	B1
Y'_3	B2
Z'_3	B3
a elliptic curve parameter	A3
b elliptic curve parameter	B0

All of the input/output numbers are expected to be in Montgomery residue system, not standard format.

The point multiply works with a minimum modulus size of 5, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 32.

```
STATUS mpc180EccDoubleQ
(
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **k** < 5
  - **modSize** < 5
  - **modSize** > 32
  - no response from the MPC180

3.2.4 mpc180EccModularAdd ()

This function performs an ECC modular addition, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N. The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

B1 = A2 + B1 mod N3

The parameter **basis** specifies if the modular addition is to be done in the F_p basis (basis = PKHA_F_p) or the F_2m basis (basis = PKHA_F_2m).

The modular addition works with a minimum of 4 digits, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 4 and 32.

```
STATUS mpc180EccModularAdd
(
unsigned int regAsel,
unsigned int regBsel
unsigned int regNsel,
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **modSize** < 4
  - **modSize** > 32
  - no response from the MPC180

3.2.5 mpc180EccModularSubtract()

This function performs an ECC modular subtraction, in F_p, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N. The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter.

For example, if **regAsel** = 0, **regBsel** = 3, and **regNsel** = 2, the calculation would be as follows:

$$B3 = A0 – B3 \bmod N2$$

The parameter **basis** specifies if the modular subtraction is to be done in the F_p basis (basis = PKHA_F_p), or the F_2m basis (basis = PKHA_F_2m).

The modular subtraction works with a minimum of 4, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 4 and 32.

```
STATUS mpc180EccModularSubtract
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel,
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **modSize** < 4
  - **modSize** > 32
  - no response from the MPC180

3.2.6 mpc180EccModularMultiply()

This function performs the Montgomery Modular Multiplication ($A \times B \times R^{-1}$) mod N, in F_p, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N, and $R = 2^{16D}$, where D is the number of digits in the modulus vector. The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

$$B1 = (A2 \times B1 \times R^{-1}) \bmod N3$$

The modular multiplication works with a minimum of 5, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) between 5 and 32.

The parameter **basis** specifies if the modular multiply is to be done in the F_p basis (basis = PKHA_F_p) or the F_2m basis (basis = PKHA_F_2m).

```
STATUS mpc180EccModularMultiply
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel,
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — **modSize** < 5
  - — **modSize** > 32
  - — no response from the MPC180

3.2.7 mpc180EccModularMultiply2()

This function performs the Montgomery Modular Multiplication.

($A \times B \times R^{-2}$) mod N, in F_p, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N, and $R = 2^{16D}$, where D is the number of digits in the modulus vector. The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

$$B1 = (A2 \times B1 \times R^{-2}) \bmod N3$$

The parameter **basis** specifies if the modular multiply is to be done in the F_p basis (basis = PKHA_F_p) or the F_2m basis (basis = PKHA_F_2m).

This function is ideal for working with affine coordinates. After a point multiply, this function may be used to exit projective coordinates. For example to find x, for $x = X/Z^2$, where X and $(Z^2)^{-1}$ are in the Montgomery residue system. Loading X and $(Z^2)^{-1}$ into the appropriate operand registers and initiating this function would yield x which is no longer in the Montgomery residue system.

The modular multiplication works with a minimum of 5 digits, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 32.

```
STATUS mpc180EccFpModularMultiply2
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel,
unsigned int basis
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — **modSize** < 5
  - — **modSize** > 32

— no response from the MPC180

3.3 MPC180PkhaRsa Interface

The following sections describe the functions that perform exponentiation on the input value, modular addition and subtraction with two vectors, and the Montgomery Modular Multiplication.

3.3.1 mpc180RsaExpA()

This function performs an exponentiation on the input value. The mathematical representation is $S = (A' \times R^{-1})^{EXP} \bmod N$.

The input parameters are expected to be loaded, using the mpc180LoadAReg and mpc180LoadNReg() functions, in the following registers.

Parameter	Register
A’	A0-3
N (modulus)	N0-3

The output parameters are left in the following registers and may be read with the function mpc180ReadBReg().

Parameter	Register
S	B0-3

The input A’ in register A0-3 should be provided in Montgomery format. If it is, the value calculated is as follows:

$$S = (A' \times R^{-1})^{EXP} \bmod N = (A \times R \times R^{-1})^{EXP} \bmod N = A^{EXP} \bmod N.$$

The parameter **exp** is the exponent, **expSize** is the size of the exponent in 32-bit words (1–64). The parameter **expLswToMsw** is a boolean which specifies the ordering of words in the input parameter **exp**. If the order of words is “LSW is lowest in memory,” **expLswToMsw** should be set to TRUE; otherwise, if the order is “MSW is lowest in memory,” it should be set to FALSE.The exponentiation works with a minimum of 5 digits, so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 128.

```
STATUS mpc180RsaExpA
(
PKHA_NUM exp,
unsigned int exposes,
BOOLEAN expLswToMsw
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  — **modSize** < 5
  — no response from the MPC180

3.3.2 mpc180RsaModularMultiply()

This function performs the Montgomery Modular Multiplication ($A \times B \times R^{-1}$) mod N, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N, and $R = 2^{16D}$, where D is the number of digits in the modulus vector. The result is returned in the B register.

The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter. Normally, these parameters are all set to 0, due to the size of RSA parameters. However, for operations such as the Chinese Remainder Theorem, they may be set to other values.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

$$B1 = (A2 \times B1 \times R^{-1}) \bmod N3$$

This routine is used to put messages into the Montgomery format.

The modular multiplication works with a minimum of 5 digits so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 128.

```
STATUS mpc180RsaModularMultiply
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — **modSize** < 5
  - — no response from the MPC180

3.3.3 mpc180RsaModularMultiply2()

This function performs the Montgomery Modular Multiplication ($A \times B \times R^{-2}$) mod N, with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N, and $R = 2^{16D}$, where D is the number of digits in the modulus vector. The result is returned in the B register.

The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter. Normally, these parameters are all set to 0, due to the size of RSA parameters. However, for operations such as the Chinese Remainder Theorem, they may be set to other values.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

$$B1 = (A2 \times B1 \times R^{-1}) \bmod N3$$

This routine may be used to take messages out of the Montgomery format.

The modular multiplication works with a minimum of 5 digits so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 5 and 128.

```
STATUS mpc180RsaModularMultiply2
```

```
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel
```

Returns one of the following:

- OK
- ERROR
  - **modSize** < 5
  - no response from the MPC180

3.3.4 mpc180RsaModularAdd()

This function performs modular addition with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N. The result is returned in the B register (B = A+B mod N).

The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter. Normally, these parameters are all set to 0, due to the size of RSA parameters. However, for operations such as the Chinese Remainder Theorem, they may be set to other values.

For example, if **regAsel** = 2, **regBsel** = 1, and **regNsel** = 3, the calculation would be as follows:

$$B1 = (A2 + B1) \bmod N3$$

The modular addition works with a minimum of 4 digits so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 4 and 128.

```
STATUS mpc180RsaModularAdd
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel
)
```

Returns: OK, or ERROR **modSize** < 4 or no response from the MPC180.

3.3.5 mpc180RsaModularSubtract()

This function performs modular subtraction with two vectors loaded into the A and B registers, where both of these vectors are less than the value stored in the modulus register N. The result is returned in the B register. (B = A-B mod N)

The parameters **regAsel** (0,1,2,3), **regBsel** (0,1,2,3), and **regNsel** (0,1,2,3) specify which of the registers to use for the operation. The result is returned in the same register as the input **B** parameter. Normally, these parameters are all set to 0, due to the size of RSA parameters. However, for operations such as the Chinese Remainder Theorem, they may be set to other values.

For example, if regAsel = 2, regBsel = 1, and regNsel = 3, the calculation would be as follows:

$$B1 = (A2 - B1) \bmod N3$$

The modular subtraction works with a minimum of 4 digits so the value of **modSize** should have been previously set (with mpc180PkhaLoadModSize) to a value between 4 and 128.

```
STATUS mpc180RsaModularSubtract
(
unsigned int regAsel,
unsigned int regBsel,
unsigned int regNsel
)
```

Returns: OK, or ERROR **modSize** < 4 or no response from the MPC180.

4 MPC180Des Module

The MPC180Des module performs high-speed encryption and decryption using the DES algorithm. It also supports two key and three key Triple-DES encryption/decryption. It can be used in both Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes of operation. This module uses DMA and the MPC180 chip's FIFO mode to transfer data for encryption/decryption.

Table 4-1. MPC180Des Module

Module	Routine
MPC180Des	mpc180DesEcb
	mpc1803DesEcb
	mpc180DesCbc
	mpc1803DesCbc

4.1 MPC180Des Routines

The following sections describe the functions that can encrypt or decrypt a buffer.

4.1.1 mpc180DesEcb()

The DES algorithm in ECB mode, together with the 64 bits of key in **key**, can be used to encrypt or decrypt a buffer (**inBuff**) of **len** bytes and returns the result in **outBuff**. If **encrypt** is true, the function will encrypt the data; otherwise, it will decrypt the data.

```
STATUS mpc180DesEcb
(
unsigned long key[2],
unsigned long *inBuff,
unsigned int len,
unsigned long *outBuff,
BOOLEAN encrypt
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — no response from MPC180
  - — **len** is not a multiple of the DES block size (8)

4.1.2 mpc1803DesEcb

The Triple DES algorithm in ECB mode, together with the three 64-bit keys in **key**, can be used to encrypt or decrypt a buffer (**inBuff**) of **len** bytes and returns the result in **outBuff**. If encrypt is true, the function will encrypt the data; otherwise it will decrypt the data.

```
STATUS mpc1803DesEcb
(
unsigned long key[3][2],
unsigned long *inBuff,
unsigned int len,
unsigned long *outBuff,
BOOLEAN encrypt
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — no response from MPC180
  - — **len** is not a multiple of the DES block size (8)

4.1.3 mpc180DesCbc

The DES algorithm in CBC mode, together with the 64 bits of key in **key** and 64 bits of initialization vector in **iv**, can be used to encrypt or decrypt a buffer (**inBuff**) of **len** bytes. It returns the result in **outBuff**. If encrypt is true, then the function will encrypt the data; otherwise it will decrypt the data.

```
STATUS mpc180DesCbc
(
unsigned long key[2],
unsigned long iv[2],
unsigned long *inBuff,
unsigned int len,
unsigned long *outBuff,
BOOLEAN encrypt
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — no response from MPC180
  - — **len** is not a multiple of the DES block size (8)

4.1.4 mpc1803DesCbc

The Triple DES algorithm in CBC mode, together with the three 64-bit keys in **key** and 64 bits of initialization vector in **iv**, can be used to encrypt or decrypt a buffer (**inBuff**) of **len** bytes. It returns the result in **outBuff**. If encrypt is true, then the function will encrypt the data; otherwise it will decrypt the data.

```
STATUS mpc1803DesCbc
(
unsigned long key[3][2],
unsigned long iv[2],
unsigned long *inBuff,
unsigned int len,
unsigned long *outBuff,
BOOLEAN encrypt
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - — no response from MPC180
  - — **len** is not a multiple of the DES block size (8)

5 MPC180Mdha Module

The MPC180Mdha module is capable of performing SHA-1, MD4, and MD5, three of the most popular public Message Digest algorithms. It is also cable of generating an HMAC as specified in RFC 2104. The HMAC can be built upon any of the hash functions supported by MPC180Mdha.

Table 5-1. MPC180Mdha Module

Module	Routine
MPC180Mdha	mpc180MdhaHash
	mpc180MdhaHmac
	mpc180MdhaHashInit
	mpc1803MdhaHmacInit
	mpc1803MdhaUpdate
	mpc1803MdhaFinal

The interfaces to MPC180Mdha are broken up into two groups. The first is intended to process complete messages through the hash or HMAC. This is accomplished by calling mpc180Hash() or mpc180Hmac(). This type of sequence is illustrated by the following:

```
unsigned char *myMsg = “This is the message to be hashed...”;
unsigned char digest[4];

mpc180Hash(MDHA_MD5, myMsg, strlen(myMsg), digest);
```

The second interface is used when the message to be hashed or HMACed is presented in chunks. This is accomplished by calling mpc180HashInit() or mpc180HmacInit() to start the hash, mpc180MdhaUpdate() to hash a chunk, and mpc180MdhaFinal() to finish the hash and store the digest. This type of sequence is illustrated by the following:

```
unsigned char *myMsg1 = “This is the message to be hashed......”;
unsigned char *myMsg2 = “And it comes in several parts......”;
unsigned char *myMsg3 = “Which must be hashed separately......”;
unsigned char *key = “AbCdEfG”;
unsigned char digest[4];
```

```
MDHA_CTX *context;
context = (MDHA_CTX *) malloc(sizeof(MDHA_CTX));

mpc180HmacInit(MDHA_MD5, context, FALSE, key, 7, NULL);
mpc180MdhaUpdate(myMsg1, strlen(myMsg1), context);
mpc180MdhaUpdate(myMsg2, strlen(myMsg2), context);
mpc180MdhaUpdate(myMsg2, strlen(myMsg2), context);
mpc180MdhaFinal(digest, context);
```

The second interface may also be used to do a precomputation step, if the same key is going to be used for many HMAC calculations. To do this, the routine is called with **context** = NULL and **precalc** = FALSE, which will hash the key, and return the partial result in ***digest**. Then, to initialize the HMAC with the precomputed digest, you may call the routine again with **context !** = NULL, **precalc** = TRUE, and the precomputed digest in ***digest**. **Note**, **key**, and **keyLen** must still be provided on this call, as they are used at the end of the HMAC calculation sequence (by mpc180MdhaFinal()).This sequence, as shown in the following code, will save time over the previous one, if the same key is reused many times:

```
unsigned char *myMsg1 = “This is the message to be hashed......”;
unsigned char *myMsg2 = “And it comes in several parts......”;
unsigned char *myMsg3 = “Which must be hashed separately......”;
unsigned char *key = “AbCdEfG”;
unsigned char preDigest[4];
unsigned char digest[4];
MDHA_CTX *context;
context = (MDHA_CTX *) malloc(sizeof(MDHA_CTX));

mpc180HmacInit(MDHA_MD5, NULL, FALSE, key, 7, preDigest);
∑
∑
∑
mpc180HmacInit(MDHA_MD5, context, TRUE, key, 7, preDigest);
mpc180MdhaUpdate(myMsg1, strlen(myMsg1), context);
mpc180MdhaUpdate(myMsg2, strlen(myMsg2), context);
mpc180MdhaUpdate(myMsg2, strlen(myMsg2), context);
mpc180MdhaFinal(digest, context);
```

5.1 MPC180Mdha Routines

The following section describe the functions that compute a message digest and authentication, initialize context structure used for hashing.

5.1.1 mpc180MdhaHash()

Compute a message digest of the **len** bytes stored at **inBuff** and place it at **digest**. The parameter **algorithm** specifies which hashing algorithm to use (algorithm = MDHA_MD4, MDHA_MD5, or MDHA_SHA1). The digest buffer must be able to hold 16 bytes for MD4 and MD5 and 20 bytes for SHA-1.

```
STATUS mpc180MdhaHash
(
unsigned int algorithm,
unsigned char *inBuff,
unsigned int len,
unsigned char *digest
)
```

Returns: OK, or ERROR if no response from the MPC180.

5.1.2 mpc180MdhaHmac()

Computes the message authentication code of the len bytes at **inBuff** using the respective hash function (MD4, MD5, SHA-1) and the key which is **keyLen** bytes long. The code is stored in the code buffer digest, which must be able to hold 16 bytes for MD4 and MD5 and 20 bytes for SHA-1. The parameter **algorithm** specifies which hashing algorithm to use (algorithm = MDHA_MD4, MDHA_MD5, or MDHA_SHA1).

```
STATUS mpc180MdhaHmac
(
unsigned int alogorithm
unsigned char *key,
unsigned int keyLen,
unsigned char *inBuff,
unsigned int len,
unsigned char *digest
)
```

Returns: OK, or ERROR if no response from the MPC180.

5.1.3 mpc180MdhaHashInit ()

Initializes a context structure used for hashing a message in parts. The structure ***context** should be pre-allocated. The parameter **algorithm** specifies which hashing algorithm to use (algorithm = MDHA_MD4, MDHA_MD5, or MDHA_SHA1).

```
void mpc180HashInit
(
unsigned int alogorithm,
MDHA_CTX *context
)
```

Returns: N/A

5.1.4 mpc180MdhaHmacInit ()

Initializes a context structure used for computing a HMAC of a message in parts. The structure ***context** should be pre-allocated. The parameter **algorithm** specifies which hashing algorithm to use (algorithm = MDHA_MD4, MDHA_MD5, or MDHA_SHA1). The key used is **keyLen** bytes long.

This routine may also be used to precalculate a partial digest for a key that is reused many times. To do this, the routine is called with **context** = NULL and **precalc** = FALSE, which will hash the key and return the partial result in *digest.

Then, to initialize the HMAC with the precomputed digest, you may call the routine again with **context !** = NULL, **precalc** = TRUE, and the precomputed digest in ***digest**. Note, key and keyLen must still be provided on this call, as they are used at the end of the HMAC calculation sequence (by mpc180MdhaFinal()).

NOTE

If **context !** = NULL and **precalc** = FALSE, then digest is ignored, and may be NULL.

```
STATUS mpc180HmacInit
(
unsigned int alogorithm,
MDHA_CTX *context
BOOLEAN precalc,
unsigned char *key,
unsigned int keyLen,
const unsigned char *digest
)
```

Returns: OK, or ERROR if no response from MPC180.

5.1.5 mpc180MdhaUpdate()

This function is called repeatedly with chunks of message data to be hashed. The data to be hashed in len bytes at **inBuff**, with the hash context of context used. The digest is computed using the hashing algorithm that was selected when either mpc180MdhHashInit() or mpc180MdhaHmacInit() was called for this context.

```
STATUS mpc180HashUpdate
(
MDHA_CTX *context,
const unsigned char *inBuff,
unsigned int len
)
```

Returns: OK, or ERROR if no response from MPC180.

5.1.6 mpc180MdhaFinal()

Places the completed message digest in **digest**, which must be able to hold 16 bytes of data for the MD4 and MD5 algorithms, and 20 bytes of data for the SHA-1 algorithm. It also erases the hash context structure ***context**. The digest is computed using the hashing algorithm that was selected when either mpc180MdhHashInit() or mpc180MdhaHmacInit() was called for this **context**.

```
STATUS mpc180HashFinal
(
unsigned char *digest,
MDHA_CTX *context
)
```

Returns: OK, or ERROR if no response from MPC180.

6 MPC180Afha Module

The AFHA Module accelerates an algorithm compatible with the RC4 stream cipher from RSA Security, Inc. The RC4 algorithm is byte oriented, therefore a byte of plaintext is encrypted with a key to produce a byte of ciphertext. The key is variable length, and MPC180Afha supports keys ranging from 40 bits to 128 bits.

Table 6-1. MPCAfha Module

Module	Routine
MPC180Afha	mpc180AfhaSetKey
	mpc180AfhaProcess

The MPC180Afha module is used by making one call to set the key for an encryption/decryption sequence, calling the process function to process sequential encryption/decryption blocks. This is illustrated in the following code segment:

```
     unsigned char *myKey = "AbCdE";
     unsigned char *block1 = "Now is the time ";
     unsigned char *block2 = "for every ";
     unsigned char *encryptBlock1;
     unsigned char *encryptBlock2;
     AFHA_CTX *context;
     context = (AFHA_CTX *) malloc(sizeof(AFHA_CTX));

     mpc180AfhaSetKey(myKey, 5, context);
     encryptBlock1 = malloc(strlen(block1));
     encryptBlock2 = malloc(strlen(block2));
     mpc180AfhaProcess(block1, strlen(block1), encryptBlock1,context);
     mpc180AfhaProcess(block2, strlen(block2), encryptBlock2,context);
```

The MPC180Afha module also is able to support encryption context switching, where a encryption/decryption sequence is temporarily suspended to allow encryption of another message with a different key. The encryption context switch is done automatically by calling mpc180AfhaSetKey() or mpc180AfhaProcess() for the second (or third, etc.) set of data. Note, however, that the context switch time can be quite large as a large amount of data must be transferred to/from the MPC180 to perform the context switch. This sequence of operations is illustrated in the following code segment:

```
unsigned char *myKey1 = "AbCdE";
unsigned char *myKey2 = "FgHiJk";
unsigned char *block11 = "Now is the time ";
unsigned char *block12 = "for every ";
unsigned char *block21 = "Four score and ";
unsigned char *block22 = "twenty years ago ";
unsigned char *encryptBlock11;
unsigned char *encryptBlock12;
unsigned char *encryptBlock21;
unsigned char *encryptBlock22;
AFHA_CTX *context;
AFHA_CTX *context2;
context1 = (AFHA_CTX *) malloc(sizeof(AFHA_CTX));
context2 = (AFHA_CTX *) malloc(sizeof(AFHA_CTX);

mpc180AfhaSetKey(myKey1, 5, context1);
encryptBlock1 = malloc(strlen(block1));
mpc180AfhaProcess(block11, strlen(block11), encryptBlock11,context1);

mpc180AfhaSetKey(myKey2, 6, context2);
```

```
encryptBlock21 = malloc(strlen(block21));
mpc180AfhaProcess(block21, strlen(block21), encryptBlock21, context2);
encryptBlock22 = malloc(strlen(block22));
mpc180AfhaProcess(block22, strlen(block22), encryptBlock22, context2);

encryptBlock12 = malloc(strlen(block12));
mpc180AfhaProcess(block12, strlen(block12), encryptBlock12, context1);
```

6.1 MPC180Afha Routines

The following sections describe different functional routines that can allow context switching, and encrypting or decrypting.

6.1.1 mpc180AfhaSetKey()

Sets up the AFHA key for use, using the **keyLen** bytes long key at **key**. Valid values for **keyLen** are between 5 and 16. The context structure pointer that is passed in is to allow for context switches to occur. If the parameter, **doNotSaveContext** is TRUE, the AFHA context which the MPC180 was currently processing will not be saved, and the old context will be invalidated.

```
STATUS mpc180AfhaSetKey
(
unsigned char *key,
unsigned int keyLen,
AFHA_CTX *context,
BOOLEAN doNotSaveContext
)
```

Returns one of the following:

- OK
- ERROR if one of the following:
  - **key** < 5
  - **key** > 16
  - no response from MPC180

6.1.2 mpc180AfhaProcess()

The **len** bytes of data at **inBuff** can be encrypted or decrypted using **key** (from mpc180AfhaSetKey()). The results are stored at **outBuff**.

```
STATUS mpc180AfhaProcess
(
unsigned char *inBuff,
unsigned int len,
unsigned char *outBuff,
AFHA_CTX *context
)
```

Returns: OK, or ERROR if no response from MPC180.

7 MPC180Rng Module

The MPC180Rng module provides the application software with a method of generating 32-bit random numbers. There are two interfaces provided, one to provide one random number, and the second to provide a block of random numbers. The single random number interface uses the MPC180 in Open Address mode, while the block random number interface employees FIFO mode and DMA.

Table 1. MPC180Rng Module

Module	Routine
MPC180Rng	mpc180Rand
	mpc180RandLongs

7.1 MPC180Rng Routines

The following sections describe the functions that generate 32-bit random numbers and store them in a buffer.

7.1.1 mpc180Rand()

This routine generates a 32 bit random number.

```
unsigned long mpc180Rand
(
)
```

Returns: 32-bit random number

7.1.2 mpc180RandLongs()

This routine generates **len** 32-bit random numbers and stores them at **buff**.

```
STATUS mpc180Rand
(
unsigned long *buff,
unsigned int len
)
```

Returns: OK or ERROR if no response from the MPC180.

8 MPC180 Interface Module Porting

The following sections describe main operating system specific concept of semaphores, the required files for the interface module code, configuration of the interface modules, and external variables and functions.

8.1 Include Files

The interface module code uses the file vxWorks.h. This includes standard vxWorks type includes. Using this code in another system will require redefining those types for that system. For example, the type UINT32 should be defined as a 32-bit unsigned integer on your target system.

In addition, for the semaphore mechanism (described in Section 8.2), the file **semLib.h** is included, to provide definitions for the semaphore type (SEM_ID) and the semaphore function calls. This will need to be replaced by the appropriate files depending on the communication mechanism used for the port.

Several drivers also use the ANSI standard string functions specified in **string.h**.

8.2 Semaphores

The main operating system specific concepts used in the code are two types of semaphores. The first of these is the mpc180Mutex, which is used to make sure that only one task is able to use the MPC180 at one time. There are three calls involved with this semaphore:

- semMCreate(mpc180Mutex….—This call creates the semaphore for later use.
- semTake(mpc180Mutex, WAIT_FOREVER)—This call takes the semaphore, preventing another task from using the MPC180.
- semGive(mpc180Mutex)—This call gives the semaphore, allowing another task to use the MPC180.

The second type of semaphores are Binary Semaphores that are used for communication between the interrupt service routine and the main line routines. In other multitasking operating systems, these calls should be replaced with the appropriate semaphore calls for the specific operating system. In a single threaded system (no multitasking OS), the Mutex is not necessary, and the Binary Semaphore mechanism may be replaced by the setting and clearing of global flags. The semaphores used in the code are as follows:

```
SEM_ID mpc180Mutex;
SEM_ID mpc180PkhaDoneSemaphore;
SEM_ID mpc180PkhaErdySemaphore;
SEM_ID mpc180DesDoneSemaphore;
SEM_ID mpc180AfhaInitSemaphore;
SEM_ID mpc180AfhaPermuteSemaphore;
SEM_ID mpc180AfhaFullMessageSemaphore;
SEM_ID mpc180AfhaSubMessageSemaphore;
SEM_ID mpc180MdhaSemaphore;
SEM_ID mpc180IdmaOutSemaphore;
```

There are three calls involved with these semaphores:

- semBCreate—This call creates the semaphore for later use.
- semTake—This call is used in mainline code to wait on the semaphore to be given in the Interrupt Service Routine. It usually has a timeout associated with it.
- semGive—This call gives the semaphore from the ISR, allowing the mainline task to continue.

Also note, if a user is chaining several of the PKHA operations to do a higher level arithmetic function, they should use the “mpc180 Mutex” in the higher level functions to prevent another task from disturbing the state of the PKHA registers between the low level function calls.

8.3 Conditional Compilation

Three configurations of the interface modules may be built:

1. Processor Transfer of Data to MPC180, MPC180 byte swapping.
   In this configuration, the processor transfers data to and from the MPC180’s FIFO buffers, not the DMA controller. In addition, byte swapping required by the algorithms is performed by switching the MPC180 endian pin by means of an externally provided function (see section 8.4). For this configuration, no macro definitions are necessary.
2. DMA Transfer of Data to MPC180, MPC180 byte swapping
   This works the same as configuration 1, except in this case, the 8260 DMA engine is used to transfer data to/from the MPC180. The required macros for this configuration are:
   #define INCLUDE_MPC180_DMA
3. Processor Transfer of Data to MPC180, Processor byte swapping.
   This is the same as configuration 1; however, the processor performs the byte swapping in software, and the external endian pin function should not be provided. The required macros for this configuration are:
   #define INCLUDE_MPC180_SW_BYTE_SWAP

NOTE

The macros for options 2 and 3 must not be defined at the same time because the drivers will not perform the required functions if they are.

8.4 Required Externals

The MPC180 interface modules require three external variables (shown in Table 8-1) and three external functions.

Table 8-1. External Variables Required by the MPC180 Interface

External variable code	Definition
extern const UINT32 MPC180_BASE_ADRS;	Specifies the base address of the MPC180 in memory.
extern const int MPC180_INPUT_DMA_CHANNEL;	Specifies which DMA channel is used to transfer data to the MPC180.
extern const int MPC180_OUTPUT_DMA_CHANNEL;	Specifies which DMA channel is used to transfer data to the MPC180.

The first of the three required external functions is as follows:

```
void sysMpc180Endian(int endian)
```

where endian is either MPC180_BIG_ENDIAN = 1 or MPC180_LITTLE_ENDIAN = 2.

This function is used to change the state of the MPC180 Endian pin and is dependent on hardware. In the reference design, this pin is tied to Parallel Port A, pin 1 on both the MPC8260 and the MPC860. The function has the following form:

```
void sysMpc180Endian
    (
    int endian
```

```
            )
            {
            int immrVal = vxImmrGet() ;
            if (endian == MPC180_BIG_ENDIAN)
                   {
                   *PDATA( immrVal ) |= PA1; /* Set to big endian */
                   }
              else
                   {
                   *PDATA( immrVal ) &= ~PA1; /* Set to little
        endian */
                   }
            }
```

The other two required external functions are calls to the DMA drivers to start DMA transfers. They have the following prototypes:

```
void ppc8260IdmaStart(int channel, unsigned char *source,
    unsigned char *destination,
                UINT32 size, BOOL interrupt, int direction);
void ppc8260IdmaChainStart(int channel, unsigned char
*source1,
    unsigned char *source2,
    unsigned char *destination,
    UINT32 size1, UINT32 size2, BOOL interrupt);
```

9 IPsec and IKE

This section describes MPC180 functionality with Wind River's modifications to the WindNet™ IPsec and WindNet IKE (these modifications are not shown in the user's guide for the base WindNet IPsec and WindNet IKE products).

These release notes support Wind River's modifications to the WindNet IPsec and WindNet IKE products to allow the use of the MPC180 Security Processor with these products. The user's guide for the base WindNet IPsec and WindNet IKE products which describes the functionality is unchanged do to these modifications. For information on the drivers used by these modifications, consult the mpc180 Interface Library Guide.

NOTE

Forward slashes are used as pathname delimiters for both UNIX and Windows filenames since this is the default for vxWorks.

9.1 Installation

The installation instructions for the base WindNet IPsec and WindNet IKE products should be followed as stated in the release notes for those products. However, before these products are built (as shown in section 2.3.2 and 2.4.2 of the IPsec/IKE release notes), the modifications for the MPC180 should be installed. If the IPsec and IKE products have been built already, the current build should be deleted as described in the IPsec/IKE release notes.

Also, the MPC180 Interface Modules must be installed prior to building of IPsec/IKE.

To install the modifications for the MPC180, extract the zip file containing the modifications (ipsec_ike_mpc180.zip) into the Tornado installation directory. Once the modifications are installed, WindNet IPSec and WindNet IKE may be built and used per their installation instructions.

9.2 Distribution Archive

For this release, the distribution archive consists of the source files listed in this section. Note that the file paths are relative to installDir/target/.

```
WNCrypto Files
src/wrn/wncrypto/openssl/sha.h
src/wrn/wncrypto/openssl/crypto/sha/Makefile
src/wrn/wncrypto/openssl/crypto/wrn/Makefile
src/wrn/wncrypto/openssl/crypto/wrn/crypto_functions_interface.h
src/wrn/wncrypto/openssl/crypto/wrn/crypto_functions_interface.c
src/wrn/wncrypto/openssl/crypto/wrn/math_functions_interface.c
src/wrn/wncrypto/openssl/crypto/wrn/md5.h
```

10 vxWorks

The following sections describe the installation of the MPC180 interface modules, BSP integration, and distribution archives.

10.1 Introduction

These release notes support MPC180 Security Processor Interface Modules for use with vxWorks. For information on the use of the interface modules (including porting issues), consult the MPC180 Interface Library Guide.

NOTE

Forward slashes are used as pathname delimiters for both UNIX and Windows filenames since this is the default for vxWorks.

10.2 Installation

To install the MPC180 Interface Modules, extract the zip file containing the source files (mpc180_drivers.zip) into the Tornado installation directory.

Once the modules are installed, the vxWorks image may be built per the following instructions.

10.3 Building the Interface Modules

Throughout the remainder of the installation instructions, the following variables are used:

Variable	Definition
cpuFamily	Specifies the target CPU family, such as PPCEC603 or PPC860
toolChain	Specifies the tool, such as **gnu**

Follow these instructions to build WindNet IPSec.

1. Go to the command prompt or shell
2. Execute **torVars** to set up the Tornado command line build environment.
3. Run **make** in the installDir/**target**/**src**/**drv**/**crypto** directory by typing these command lines switches:
   make CPU = cpuFamily TOOL = toolChain

The command line in step 3 builds the configuration Processor Transfer of Data to MPC180, MPC180 byte swapping (No macro definitions). To build the other configurations the following commands should be used instead of step 3.

1. DMA Transfer of Data to mpc180, mpc180 byte swapping (#define INCLUDE_MPC180_DMA)
   make CPU = cpuFamily TOOL = toolChain DMA = YES
2. Processor Transfer of Data to mpc180, Processor byte swapping. (#define INCLUDE_MPC180_SW_BYTE_SWAP)
   make CPU = cpuFamily TOOL = toolChain SWBYTESWAP = YES

10.4 BSP Integration

Once the modules are built, they may be integrated with the users board support package.

10.5 Distribution Archive

For this release, the distribution archive consists of the source files listed in this section. Note that the file paths are relative to installDir/target/.

```
h/drv/crypto/mpc180.h
h/drv/crypto/mpc180Afha.h
h/drv/crypto/mpc180Des.h
h/drv/crypto/mpc180Dma.h
h/drv/crypto/mpc180Init.h
h/drv/crypto/mpc180Mdha.h
h/drv/crypto/mpc180Pkha.h
h/drv/crypto/mpc180Rng.h
h/drv/dma/ppc8260Idma.h
src/drv/crypto/Makefile
src/drv/crypto/mpc180Afha.c
src/drv/crypto/mpc180Des.c
src/drv/crypto/mpc180Dma.c
src/drv/crypto/mpc180Init.c
src/drv/crypto/mpc180Mdha.c
src/drv/crypto/mpc180Pkha.c
src/drv/crypto/mpc180Rng.c
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140
(800) 441-2447

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong
852-26668334

TECHNICAL INFORMATION CENTER:

(800) 521-6274

HOME PAGE:

www.motorola.com/semiconductors

MPC180SWUG/D