MosChip Security Processor MCS1000

Software Application Note

Version 2.0

March 9, 2004

For more information contact the Technical Marketing Group:
info@moschip.com (408) 737-7141 x124
www.moschip.com

Copyright © 2004 MosChip Technologies, Inc. All Rights Reserved.

All information in this document is believed to be accurate as of the publish date.

VPNow™ is a registered trademark of Artec Group. ARM is a registered trademark of ARM Limited. All other brands or product names are the property of their respective holders.

MosChip Semiconductor products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of MosChip Semiconductor, Ltd.

MosChip Semiconductor believes the information in this document to be accurate and reliable. However, it is subject to change without notice. No responsibility is assumed by MosChip Semiconductor for its use, nor for infringement of patent or other rights of third parties. No part of this document may be reproduced, or transmitted in any form or by any means without prior consent of MosChip Semiconductor, Ltd.

Copyright © 2004 MosChip Semiconductor All Rights Reserved.

Scope
This application note illustrates the software routines that interface with the MCS1000 specific hardware. This is intended to be used by the Linux developer as well as the RTOS developer.

Introduction
The software that is included with the MCS1000 Development Board consists of two parts. Boot-loader and a Linux port that runs FreeS/WAN. This document discusses and give examples for general OS and hardware issues such as the memory and interrupt controllers, timers, serial port, and Ethernet interfaces.

MCS1000 Hardware Setup Examples

This following gives an overview and examples of how the various interfaces and internal hardware of the MCS1000 can be configured. The code examples are taken from the MCS1000 ARM Linux and from the ARMBoot port.

Memory controller
Most OS images use a code relocation technique in order to boot the system. Normally a small relocatable piece of code is provided with an image that configures the memory controller, copies application code into correct location, and executes it. The MCS1000 comes out of reset with the memory controller setup with the first region configured for on-chip ROM and the last region configured for 8-bit external EEPROM. The MCS1000 first boots from the internal ROM. The first instruction jumps to the last memory region with offset 0x100 – that is, to address 0x1C000100. Bootstrap BS3 (SW4 of S1 on the MCS100 Development Board) controls whether the MCS100 boots from its internal ROM or and external device. If SW4 is off, the MCS1000 will boot from the internal ROM and if SW4 is on the MCS1000 will boot from an external device. The offset 0x100 must be taken into account when building applications for external EEPROM. In order to copy code into memory, the loader must set up the memory controller. See MCS1000 Programmer's Reference for more details. The following main steps must be taken:

a. Switch off default configuration (see Programmer's Reference, System Configuration Register)
    - Clear register offset +0x1C bit 0 to use physical memory at address 0x00000000
b. Set up required memory region
    - On development board the following regions are available:
        - MR_CSX0 32Mbyte SDRAM 48LC8M16A2
        - MR_CSX1 (0x04000000), ADC_CSX1 (0x14000000), and ADC_SCX2 (0x18000000) for PSRAM

- ADC_CSX0 (0x10000000) for 8Mbyte Intel StrataFlash (28F640)
- ADC_CSX3 (0x1C000000) for EEPROM

- For SDRAM's use the following configuration (NO = 0):
  The following table describes SDRAM configuration in normal conditions. See a* for FIB'ed configuration.

Memory Controller configuration register offset	Value
(NO+1)*256 + 0x00	0x04017214
(NO+1)*256 + 0x04	0x00000000
(NO+1)*256 + 0x08	0x00000000
(NO+1)*256 + 0x0C	0x00000000
(NO+1)*256 + 0x10	0x00000000
(NO+1)*256 + 0x14	0x00000000
(NO+1)*256 + 0x18	0x00000000
(NO+1)*256 + 0x1C	0x00000000
(NO+1)*256 + 0x20	0x00000000
(NO+1)*256 + 0x24	0x00000000
(NO+1)*256 + 0x28	0x00000572

a*: On FIB'ed versions, the 12'th bit, the Reset Region bit, of register 0x28 is connected directly to the SDRAM controller internal signal that determines the initialized state. The SDRAM requires initialization after power up. This will be done automatically on the first access to the required region. The initialized state determines if this region is already initialized. **In FIB'ed versions the initialization has to be done manually.** This is a very complicated task and should be carried out using the initialization code provided by MosChip.

- For PSRAM's use the following configuration (NO is the region number – 1 for MR_CSX1, 5 for ADC_CSX1, 6 for ADC_CSX2):

Memory Controller configuration register offset	Value
(NO+1)*256 + 0x00	0x06003212
(NO+1)*256 + 0x04	0x00000000
(NO+1)*256 + 0x08	0x00007F80
(NO+1)*256 + 0x0C	0x00007F00
(NO+1)*256 + 0x10	0x00007F00

(NO+1)*256 + 0x14	0x00007F00
(NO+1)*256 + 0x18	0x00007F00
(NO+1)*256 + 0x1C	0x00000080
(NO+1)*256 + 0x20	0x00000000
(NO+1)*256 + 0x24	0x00000000
(NO+1)*256 + 0x28	0x00000000

- For Intel Strata Flash use the following Memory Controller configuration: (NO = 4 for ADC_CS0)

Memory Controller configuration register offset	**Value**
(NO+1)*256 + 0x00	0x08023112
(NO+1)*256 + 0x04	0x00000000
(NO+1)*256 + 0x08	0x00007E00
(NO+1)*256 + 0x0C	0x00001C00
(NO+1)*256 + 0x10	0x00001C00
(NO+1)*256 + 0x14	0x00007C00
(NO+1)*256 + 0x18	0x00003C00
(NO+1)*256 + 0x1C	0x00000400
(NO+1)*256 + 0x20	0x00000000
(NO+1)*256 + 0x24	0x00000000
(NO+1)*256 + 0x28	0x00000000

Interrupt Controller
In general the interrupt controller needs to be set-up based on the requirements of the RTOS or OS being used. See Section 3.2.3 of the MCS1000 Data Sheet for detailed information on the interrupt controller.

Timer
The timer is usually used for time measurement or task switch purposes in an operating system. Therefore, the constant interval tick, typically 100 times in second, is required. The MCS1000 has a simple 32 bit counter/timer that runs from the 50MHz clock. An interrupt will be generated on overflow e.g. count changes from 0xFFFFFFFF to 0x00000000. The timer can be written to or read from at any time. To provide constant timer tick, the timer interrupt handler needs also reload a new timer preset value into the Timer Counter Register. There is no auto reload for this timer. The following code example shows timer interrupt handler in Linux:

```
#define TIMER_BASE (0x400F800C)
#define TIMER_RELOAD_VAL ((unsigned long) (0 - (50000000/HZ)))

typedef struct TimerStruct {
    unsigned long TimerValue;
```

```
     unsigned long TimerControl;
} TimerStruct_t;

static void mcs1000_timer_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
{
volatile TimerStruct_t *timer = (volatile TimerStruct_t *)
TIMER_BASE;
 __clear_irq(IRQ_TIMER);
  unsigned int ticks = timer->TimerValue;
  timer->TimerValue = ticks + TIMER_RELOAD_VAL;
  do_timer(regs);
}
```

To start the timer a "1" needs to be written to the Timer Control Register.

In order to use the timer as a cycle counter to evaluate certain code performance use the following schema to convert timer ticks to microseconds:

```
ticks1 = timer->TimerValue; //get start state

{… code …}

ticks2 = timer->TimerValue; //get end state

if (ticks2<=ticks1)
      ticks2 += (unsigned long) (0 - ticks1);
else
      ticks2 = ticks2 - ticks1;
usecs = ticks2 / 50; //convert to microseconds
```

Serial Port

The MCS1000 has a 16550 compatible serial port with 16-byte FIFO's. In most systems, the serial port is used as an output for debugging information during the system boot-up. After boot-up, the device driver takes control over the port. The device driver complexity depends on the operating system used, but in the debugging stage only a minimum functionality is needed so that printf() or put() functions to work. The following code examples demonstrate MCS1000 serial port usage while the ARMBoot boot loader is used:

```
void serial_init(bd_t *bd)
{
  int br = (50000000/16)/(bd->bi_baudrate);//CONFIG_BAUDRATE;
  wr_b(__VSPUART_REGS_LCR, LCR_DLEN);//Enable DLL/DLH registers
  wr_b(__VSPUART_REGS_DLL, br & 0xFF); //select baud
  wr_b(__VSPUART_REGS_DLH, br>>8);
  wr_b(__VSPUART_REGS_LCR, LCR_8D | //select 8bit transfer
       LCR_1S ); //1 stop bit, no parity, disable DLL/DLH
  wr_b(__VSPUART_REGS_IER, 0); //disable all interrupts
  wr_b(__VSPUART_REGS_FCR, 0); //disable fifo's
}
```

```
int serial_getc(void)
{
   while ((rd_b(__VSPUART_REGS_LSR) & LSR_Data_Ready) == 0);
   return rd_b(__VSPUART_REGS_RBR);
}

void serial_putc(const char c)
{
   while ((rd_b(__VSPUART_REGS_LSR) & LSR_Tx_Empty) == 0);
   wr_b(__VSPUART_REGS_THR, c);
   /* If \n, also do \r */
   if(c == '\n') serial_putc('\r');
}

int serial_tstc(void)
{
   return (rd_b(__VSPUART_REGS_LSR) & LSR_Data_Ready)? 1:0;
}
```

Serial EEPROM support

The serial EEPROM support on MCS1000 is provided through dedicated GPIO pins. The protocol can be implemented in software and thus can be completely application specific. The serial EEPROM utilities source files are located with the other source code.

Ethernet MAC and PHY

MCS1000 has three on-chip Ethernet MACs connected to on-chip PHYs. The configuration of each PHY is handled through MAC1. The following code examples demonstrate MAC1 usage in ARMBoot for receiving and transmitting packets. In these examples DMA polling is used instead of interrupts.

```
#define get_reg(regno) rd_d(_CONFADDR_MAC1 + regno)
#define put_reg(regno, val) wr_d(_CONFADDR_MAC1 + regno, val)
//--------------------------------------------------------------------
---------
void write_phy_reg(unsigned int phy_addr, unsigned int reg_addr,
unsigned short data)
{
    unsigned int a;
    while (get_reg(MAC_MIIADDR)&1);
    put_reg(MAC_MIIDATA,data);
    a= phy_addr<<11 | reg_addr<<6 | 0x02;
    put_reg(MAC_MIIADDR,a);
}
//--------------------------------------------------------------------
---------
unsigned short read_phy_reg(unsigned int phy_addr, unsigned int
reg_addr)
{
    unsigned int a;
```

```
    while (get_reg(MAC_MIIADDR)&1);
    a=phy_addr<<11 | reg_addr<<6 | 0x00;
    put_reg(MAC_MIIADDR,a);
    while (get_reg(MAC_MIIADDR)&1);
    return get_reg(MAC_MIIDATA);
}

volatile unsigned char buf[2048 + 10];
volatile unsigned int  rxdma_addr, txdma_addr, phy_addr, rx_buf;
volatile int link_up;

//--------------------------------------------------------------------
---------
int eth_init( bd_t *bd )
{
  unsigned char *ea = bd->bi_enetaddr;

  /* set the ethernet address */
  /* NB! Write the address in reverse order !! */
  wr_d(MAC_BASE+MAC_ADDRL, (ea[3] << 24) | (ea[2] <<16) | (ea[1]<<8)
| ea[0]);
  wr_d(MAC_BASE+MAC_ADDRH, (ea[5] <<8) | ea[4]);

  wr_d(MAC_BASE+MAC_CNTRL, 0x1010052C);  //full duplex

  switch(MAC_BASE) {
  case _CONFADDR_MAC1 :
    rxdma_addr = _CONFADDR_PUMP + 0*32;
    txdma_addr = _CONFADDR_PUMP + 2*32;
    phy_addr   = 0x11;
    wr_b(_CONFADDR_SYSDBG + 0x04,0x01); // set leds to lkact and
rx/tx mode
    break;
  case _CONFADDR_MAC2 :
    rxdma_addr = _CONFADDR_PUMP + 1*32;
    txdma_addr = _CONFADDR_PUMP + 3*32;
    phy_addr   = 0x12;
    wr_b(_CONFADDR_SYSDBG + 0x05,0x01); // set leds to lkact and
rx/tx mode
    break;
  case _CONFADDR_MAC3 :
    rxdma_addr = _CONFADDR_PUMP + 6*32;
    txdma_addr = _CONFADDR_PUMP + 7*32;
    phy_addr   = 0x13;
    wr_b(_CONFADDR_SYSDBG + 0x06,0x01); // set leds to lkact and
rx/tx mode
    break;
  default : printf("False MAC base\n"); return 0;
  }

  if ((read_phy_reg(phy_addr, 0x1) & 0x04) == 0) { //test link
status
    write_phy_reg(phy_addr,0x4, 0x01E1); //advertise all:
100FD/100HD, 10FD,10HD
```

```
    write_phy_reg(phy_addr,0x0, 0x1200); //restart autoneg
  } else {
    link_up = 1;
  }

  rx_buf = (unsigned int) &buf[0];
  rx_buf += 8-(rx_buf&(8-1)); //get the 8 byte aligned buffer

  wr_d(rxdma_addr + 0, rx_buf); //set the
  wr_d(rxdma_addr + 4, 2048-1); //max count
  wr_d(rxdma_addr +12, 1);      //start rx dma. we use double
buffering

  return 0;
}

unsigned int read_unaligned_dword(unsigned int addr) {
  unsigned char *p = (unsigned char *) addr;
  unsigned int dw;
  unsigned char *b = (unsigned char *) &dw;
  *(b+0) = *(p+0);
  *(b+1) = *(p+1);
  *(b+2) = *(p+2);
  *(b+3) = *(p+3);
  return dw;
}

int packet_ok(unsigned int status, int len) {
  //check status word bits
  if ((status & 0x3FFF) != len) return 0; //length error
  if (status & 0x80000000) return 0; //missed frame
  if (status & 0x40000000) return 1; //packet filter pass
  return 0;
}

/* Get a data block via Ethernet */
extern int eth_rx(void)
{
    int rxlen = 0;
    if ((rd_d(rxdma_addr +12)&1) == 0) { //check if dma ready
     unsigned int status;
      rxlen = rd_d(rxdma_addr +8) -4; //last 4 bytes are status word
      status = read_unaligned_dword(rx_buf + rxlen);
      if (packet_ok(status, rxlen)) {
         int i;
         rxlen-=4;
         for(i=0;i<rxlen;i++) //copy received data into right buffer
               *(((unsigned char *)NetRxPackets[0])+i) =
*(((unsigned char *)rx_buf)+i);
         NetReceive(NetRxPackets[0], rxlen);
      } else {
         rxlen = 0;
      }
```

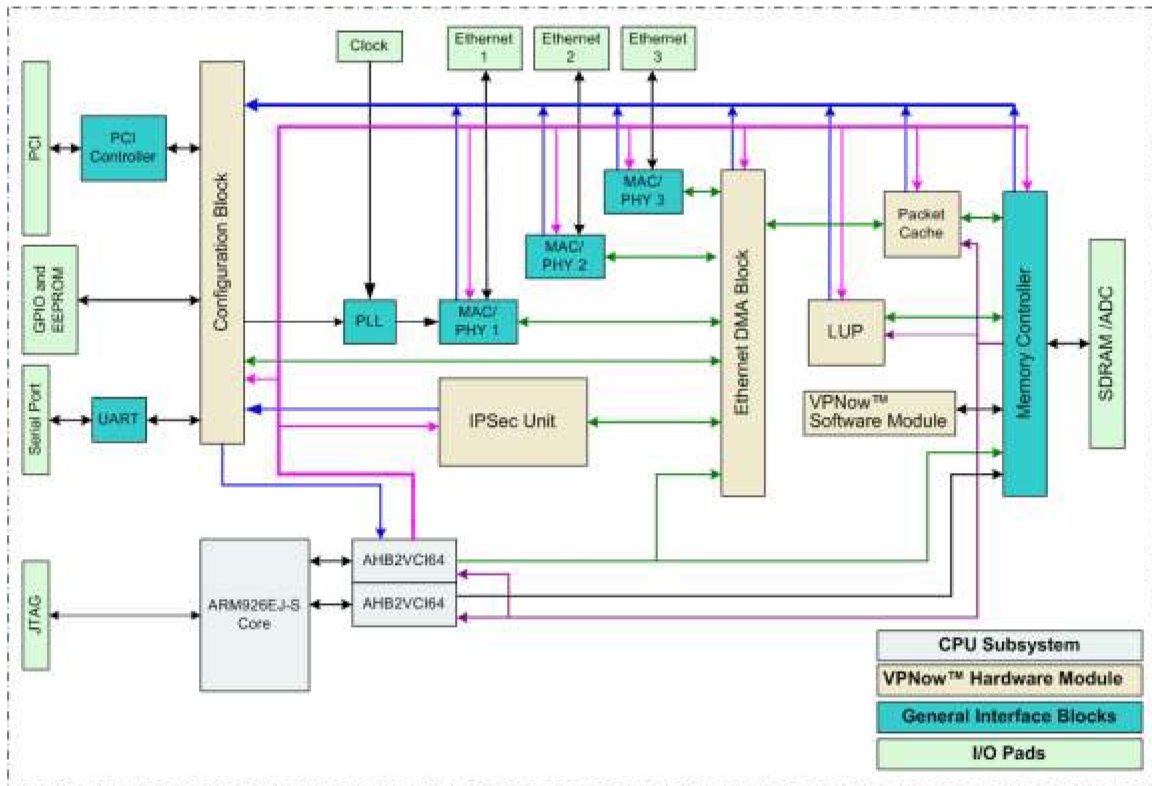
```
      wr_d(rxdma_addr +12, 1);    //restart dma, buffer address and
max count are already set
    }
    return rxlen;
}

void report_link(unsigned short d) {
  if (d&0x8000) {
    switch((d>>8)&0x7) {
    case 0: printf("No common denominator\n"); break;
    case 1: printf("10Base T\n"); break;
    case 2: printf("10Base T/Full duplex\n"); break;
    case 3: printf("100Base TX\n"); break;
    case 4: printf("100Base T4\n"); break;
    case 5: printf("100Base TX/Full duplex\n"); break;
    default: printf("not defined\n");
    }
  } else {
    printf((d&0x8)? "100MB/s\n":"10MB/s\n");
  }
}


/* Send a data block via Ethernet. */
extern int eth_send(volatile void *packet, int length)
{
    unsigned int tmo;
    unsigned int status;
    int cnt = 0;

    wr_d(txdma_addr + 0, (unsigned int) packet); //set packet start
address
    wr_d(txdma_addr + 4, length -1); //set packet lenght
    //start DMA
 retry:
    if (!link_up) {
      if ((read_phy_reg(phy_addr, 0x1) & 0x04) == 0) {
         printf("\nWaiting for LINK\n");
         while ((read_phy_reg(phy_addr, 0x1) & 0x04) == 0)
               if (ctrlc()) return 0;
      }
      printf("\nLink Up: ");
      status = read_phy_reg(phy_addr,0x19);
      wr_d(MAC_BASE+MAC_CNTRL, 0x1010052C);  //full duplex
      if ((((status>>8)&0x7) == 3) || (((status>>8)&0x7) == 1))
         wr_d(MAC_BASE+MAC_CNTRL, 0x1080052C);  //half duplex,
disable receive own
      report_link(status);
      for(tmo=0; tmo<100000; tmo++) ; //small delay after link
detected
      link_up = 1;
    }
    wr_d(txdma_addr +12, 0xD); //start transmit DMA
    /* wait for transfer to succeed */
```

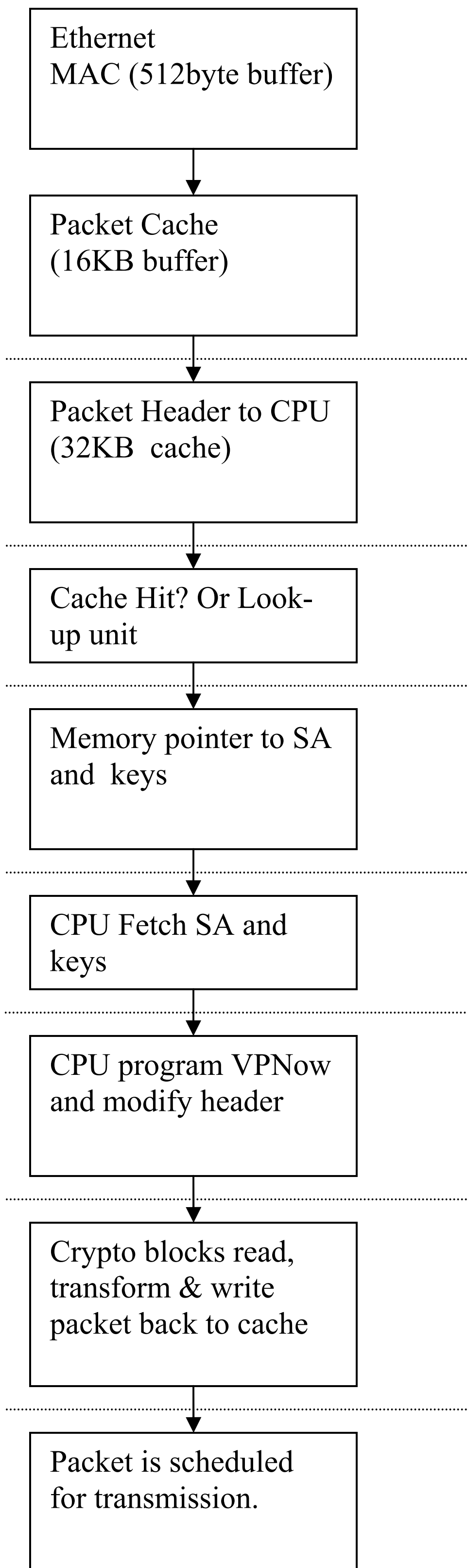
```
        tmo = get_timer(0) + 5 * CFG_HZ; //5 seccond timeout
        while(rd_d(txdma_addr +12)&1)
          if (get_timer(0) > tmo) break;

          /* nothing */ ;
        if (rd_d(txdma_addr +12)&1) {
          printf("\ntransmission error: DMA not ready\n");
          wr_d(txdma_addr +12,2); //manually stop DAM, reset FIFO
          wr_d(txdma_addr +12,0); //clear reset FIFO bit
        } else {
          status = rd_d(txdma_addr + 16);
          if ((status & 0x1) && (cnt++ < 10))
             goto retry;
        }
        return 0;
    }
```

PHY Configuration

Under normal conditions PHY needs only minimum configuration. After start-up write 0x05E1 to the PHY register 0x04. This sets auto-advertise bits to enable 100/10 full duplex/half duplex operations. The register can be written individually for separate PHY's (addresses: 0x11, 0x12, and 0x13) or to all PHY's at the same time using broadcast address 0x00. Writing 0x1200 to PHY register 0x00 restarts auto advertisement. Link status can be monitored from register 0x01 bit 2. A small delay (ca 1ms) after link pass state is required before starting data transmission. Auto negotiation result state can be monitored from register 0x19. See PHY documentation for more details.

MCS1000 Cipher Driver Data Flow Diagram
Use the diagram below as a reference to the hardware blocks in the chip. This description is intended to be an overview to provide programmers with a basic understanding of the HW architecture. The Data Flow Diagram below should be used as a reference for the MCS Cipher Driver code below.





Packet arrives at Ethernet port and loads the MAC FIFO (512B). Upon packet completion or FIFO full, interrupt to CPU allows Ethernet driver to DMA FIFO to packet cache.

The packet catch is allocated as 8 - 2KB buffers allowing enough room for the max packet size plus the header information and any modifications.

The CPU reads the header information and performs an SA look-up. It either has the data in cache or it turns on the Look-up unit.

The HW LUP searches memory for the correct SA and returns the pointer to the data via interrupt.

The CPU has the pointer to the block of memory that contains all the information required for the data processing by the VPNow engine.

The CPU fetches the transaction information via DMA from memory.

The configuration information is loaded into the memory mapped VPNow registers. The packet header is modified by the CPU and appended to the packet.

The cryptography and authentication operators are chained and the packet is shifted through the VPNow unit. Upon completion the CPU is interrupted.

The driver loads the DMA engine to transmit the packet to the Ethernet via the MAC FIFO.

MCS1000 Cipher unit driver

The Cipher unit driver executes asynchronous cryptography calls. The tasks are queued to the driver wait list. This list is scanned with cipher FIQ calls that schedule the tasks for the hardware unit. After the task is complete it is sent to the output ring buffer. If the defined number of tasks has completed then the system IRQ is set which in turn schedules callback functions as kernel threads. If the callback function is hardware IRQ driven then the thread scheduling overhead can be avoided.

To start a cryptographic session function *SCP_cipher()* should be called:

```
int SCP_cipher(
        Callback callback,
        CallbackData* callbackdata,
        CipherData* cipherdata)
```

The *callback* field specifies the function that will be called on completion of the Cipher unit processing. The *callbackdata* field is caller specific. The *cipherdata* field points to the *CipherData* structure which specifies the required algorithms, the algorithm-specific parameters, the data offsets in buffers, and the key-data. These structures are defined in the file *drivers/net/mcs1000_fiq/fiq_cipher.h.*

To improve performance the Cipher driver does not error check the parameters *callback, callbackdata and cipherdata.* It is the responsibility of the programmer to take appropriate measures to avoid system crashes. The function *SCP_cipher*() must be called with the correct parameters for cryptographic calculations such as block alignment of cipher processing start and end. Correct padding for cipher blocks must be done prior to the calling *SCP_cipher*().

The *SCP_cipher()* function will return 0 if the scheduled crypto-job has been successfully prepared for cipher unit, or a negative value if an error occurred:

Error code	Error
-1	All cipher unit resources are busy. This will happen if the input queue is full.
-1015	SCP Cipher unit does not support required authentication algorithm.
-1016	SCP Cipher unit does not support required encryption algorithm.
-1017	Algorithms are not specified.
-1018	Error in job description.

The *CipherData* structure contains the required fields for processing a packet by the cipher engine. These fields must be set by the caller to describe the cryptographic transforms to be done on given data:

```
struct CipherData
{
    unsigned char       *idata;
    unsigned char       *odata;
    unsigned char       dir;
     int                len;
    unsigned char       prot;
    long                auth_alg;
    unsigned char       auth_key[32];
    short               auth_key_len;
    unsigned long       auth_start;
    unsigned long       auth_len;
    unsigned long       icv_mode;
    unsigned long       icv_mask;
    unsigned char       icv[20];
    unsigned long       icv_start;
    short               icv_len;
    unsigned long       *masked_ip;
    short               masked_ip_len;
    unsigned long       masked_ip_start;
    unsigned long       ip_opt;
    unsigned long       *ip_options;
    unsigned long       ip_options_start;
    short               ip_options_len;
    long                encr_alg;
    unsigned char       encr_key[32];
    short               encr_key_len;
    unsigned long       encr_start;
    unsigned long       encr_len,
    unsigned long       iv_start;
    unsigned long       iv_len;
    unsigned long       callback_is_irq_safe;
} ;
```

idata
Specifies a pointer to the data buffer from where the cipher unit should get data for cryptographic calculations.

odata
Specifies the data buffer where the cipher unit should forward the encrypted data. These two can point to same region.

dir
Determines whether this packet will be decrypted or encrypted:

0 – decryption
1 -- encryption

len
The value in this field specifies the amount (in bytes) of data in the buffer *idata.* The end of the data in the buffer must be doubleword aligned.

prot
Specifies IPSec protocol

Protocol	Code
AH	0
ESP	1
AH + ESP	2

If AH and ESP are both required then only MAC for AH will be calculated. Currently only protocol ESP is supported.

auth_alg
Specifies an authentication algorithm. The cipher unit supports the following authentication algorithms:

Algorithm	Code (IANA)	Key length in double-words
MD5	1	0
SHA1	5	0
SHA2	6	0
HMAC-MD5	2	4
HMAC-SHA1	3	5
HMAC-SHA2	4	8
AUTHENTICATION_NONE	0	0

A 0 value in this field (AUTHENTICATION_NONE) specifies that authentication is not required.

auth_key
Specifies an array of key data used in authentication calculations. Key material must be in the raw format and not extended as in the FreeS/WAN database.

auth_key_len

Specifies the length of the authentication key in double-words for specified algorithm (see table above).

auth_start
Specifies offset (in bytes) of data from the start of the buffer *idata* to be authenticated.

auth_len
Specifies the amount of data (in bytes) to be authenticated.

icv_mode
Specifies if calculated signature (ICV) should be copied into the IP packet or into the separated array given in the field *icv*.

Code	Action
0	Copy calculated ICV into the buffer *icv*
1	Copy calculated ICV into the buffer *odata* starting at *icv_start*

icv_mask
Specifies if ICV area in the IP packet is masked (cleared) or not:

Code	Masking
0	ICV area in AH header of an IP packet is not masked
1	ICV area in AH header of an IP packet is masked

If set to 0 and AH is required then ICV data will be zeroed before authentication (in the case of an inbound packet original ICV will be restored after calculations).

icv
Specifies the buffer for calculated ICV if the value in the field *icv_mode* is set to 0.

icv_start
Specifies the offset (in bytes) of the calculated signature (ICV) in the buffer *odata.* Value in this filed will be not used if *icv_mode* is set to 0.

icv_len
Specifies the length of the signature in the double-words to copy into the buffer *odata* or return after calculations.

masked_ip
If protocol AH is involved in IPSec processing then some fields in the IP header must be masked before authentication. This field must be set to NULL if masking has been done in the buffer *idata* already.
A masked IP header may be given separately. In this case value in this field points to the buffer with prepared data. Masked IP header will be copied before authentication into the

buffer *idata* starting from *masked_ip_start.* The original IP header will be restored after calculations.

masked_ip_len
Specifies the length of masked IP header data (in double-words) pointed by the field *masked_ip* or starting at the offset given by *masked_ip_start*.

masked_ip_start
Specifies the offset (in bytes) of the masked IP header in the buffer *idata.*

ip_opt
FreeS/WAN skips options added to the IP header so options will be removed before authentication starts in the cipher unit. The value in this field specifies how to manage field *ip_options*:

Code	Action
0	No need to skip IP options, calculate
1	Options are masked, given separately and should be copied into the IP packet before calculating ICV. The value in the field *ip_options* points to the data array and field *ip_options_start* is relative offset into the packet data
2	Options should be skipped

ip_options
Specifies a pointer to the array of masked IP options if the value in the field *ip_opt* is 1.

ip_options_start
Specifies the offset (in bytes) where options of the IP header start in the buffer *idata*.

ip_options_len
Specifies the amount of options (in bytes) added to the IP header.

encr_alg
Specifies required encryption algorithm. SCP Cipher unit supports following encryption algorithms:

Algorithm	Code (IANA)	Length of the key in double-words	Length of the IV in double-words
ENCRYPTION_NONE	0	0	0
DES	2	2	2
3DES	3	6	2
AES128	12	4	4
AES192	12	6	4
AES256	12	8	4

NULL_ENCRYPTION	11	0	0

No crypto-calculations will be done if this field is set to ENCRYPTION_NONE or NULL_ENCRYPTION.

encr_key
Specifies an array of key data used in encryption calculations. Key material must be in the raw format not in the scheduled format as in the FreeS/WAN database.

encr_key_len
Specifies the length of the encryption key in the double-words. (see table above).

encr_start
Specifies offset (in bytes) of data in the buffer *idata* to be encrypted/decrypted.

encr_len
Specifies the amount (in bytes) of data to be encrypted/decrypted.

iv_start
Specifies the offset (in bytes) of the initial vector (IV) for the encryption algorithm in the buffer *idata.*

iv_len
Specifies the length (in double-words) of the IV for specified encryption algorithm (see table above).

callback_is_irq_safe
Avoid kernel thread scheduling in callback (1). Use only when the callback function is hard IRQ safe. Default value of 0 schedules callback function as kernel thread, this adds software scheduler latency to callback function.

All data-offsets must be given relative to the start of the data buffer *idata.*
If authentication is not required (*auth_alg* is AUTHENTICATION_NONE) the values in the fields *auth_key, auth_key_len, auth_start, auth_len, icv_mode, icv_mask, icv, icv_start, icv_len, masked_ip, masked_ip_len, masked_ip_start, ip_opt, ip_options, ip_options_start,* and *ip_options_len* are not used.
If encryption is not required (*encr_alg* is ENCRYPTION_NONE or NULL_ENCRYPTION) the values in the fields *encr_key, encr_key_len, encr_start, encr_len, iv_start,* and *iv_len* are not used.

If protocol AH is involved in the IPSec processing then some fields in the IP packet must be cleared (masked) before applying an authentication algorithm. In this case the parameters *masked_ip, masked_ip_len, masked_ip_start, ip_opt, ip_options, ip_options_start,* and *ip_options_len* should describe the situation properly.

The required transforms will be applied to the data from the buffer *idata*. If encryption/ decryption is involved and *idata* and *odata* point to the different buffers then encrypted data will be forwarded into the buffer *odata* and original data in the buffer *idata* will be unchanged. Original data will be replaced by decrypted data if *idata* and *odata* point to the same buffer.

ICV will be copied into the IP packet data starting at *icv_start* byte relative to the start of the buffer *odata* (original data leaves unchanged if *idata* and *odata* differ) if the value in the field *icv_mode* is set to 1, or into the separated array *icv* if *icv_mode* is set to 0.

The *callback* function will be called after the cipher engine has determined that the cipher processing for corresponding *cipherdata* is complete. The status flag indicates whether the processing was successful (0) or not (-1). Depending on the value of **callback_is_irq_safe** switch the HW IRQ function is either executed directly in interrupt context or scheduled as a kernel thread (soft interrupt, formerly also known as bottom-half). If callback does use only hard IRQ-safe functions then latencies can be avoided by setting control flag to 1.

> **Note. Be sure to check the code thoroughly because a failure can manifest itself in different areas of the kernel not directly related to the cipher call. The behavior is set once for each HW IRQ call so if IRQ-safe and non IRQ-safe calls are mixed then the first non IRQ-safe call in one system IRQ call will set all other tasks within same IRQ call to soft interrupt. The added delay is insignificant for a system that is not heavily loaded system but as the resources become more taxed the latencies will increase. It is up to the programmer to decide whether to change the context from HW IRQ to SW IRQ.**

ARM DSP Functions
The ARM926EJ-S has some DSP functionality. This functionality can be implemented using assembler macros along with the GNU Linux tools. The macros are required because the GCC does not self-invoke the gcc assembler instructions. These can be in separate assembler (.s) file or as a c macros. The following code example shows how to use qdadd instruction = result = arg1 + (arg2*2).

The code below is compiled with cross-compiler (mcs1000-gcc):

```
mcs1000-gcc dsp_asi.c -o dsp_asi
```

This command line creates a shared 32-bit arm elf executable.

There is a document describing the use of DSP instructions on the ARM Website:
http://www.arm.com/armtech/5DTJ4U/$File/DSPExt.pdf

ARM DSP Page:
http://www.arm.com/armtech/DSP?OpenDocument

Below are two versions of the code produced by gcc. If the gcc is used without the -O3 (inline functions) then the result is thirteen memory moves and one DSP instruction in witch the speed of the DSP instructions are lost in memory handling. In the second example the -O3 switch is used and the function is inline and significantly more efficient.

```
Source file: dsp_qdadd.c

unsigned int qdadd(unsigned int px, unsigned int py)
{
  unsigned int ret;
  asm volatile("qdadd r3,%0,%1":
            "=r" (ret):
            "r" (py), "r" (px)
            );
return ret;
}

int   main(int argc, char *argv[])
      {
        unsigned int a,b,tagasi;
        a=192;
        b=113;
        tagasi = qdadd(a,b);
        printf("saime %d",tagasi);
      }
```

```
Produced assembler file without inline code: dsp_qdadd.s:

      .file "dsp_qdadd.c"
      .text
      .align      2
      .global     qdadd
      .type qdadd, %function
qdadd:
      @ args = 0, pretend = 0, frame = 12
      @ frame_needed = 1, uses_anonymous_args = 0
      mov   ip, sp
      stmfd sp!, {fp, ip, lr, pc}
      sub   fp, ip, #4
      sub   sp, sp, #12
      str   r0, [fp, #-16]
      str   r1, [fp, #-20]
      ldr   r2, [fp, #-20]
      ldr   r3, [fp, #-16]
      qdadd r3,r3,r2
      str   r3, [fp, #-24]
      ldr   r3, [fp, #-24]
```

```
        mov     r0, r3
        ldmea fp, {fp, sp, pc}
        .size qdadd, .-qdadd
        .section        .rodata
        .align          2
.LC0:
        .ascii          "saime %d\000"
        .text
        .align          2
        .global         main
        .type main, %function
main:
        @ args = 0, pretend = 0, frame = 20
        @ frame_needed = 1, uses_anonymous_args = 0
        mov     ip, sp
        stmfd sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #20
        str     r0, [fp, #-16]
        str     r1, [fp, #-20]
        mov     r3, #192
        str     r3, [fp, #-24]
        mov     r3, #113
        str     r3, [fp, #-28]
        ldr     r0, [fp, #-24]
        ldr     r1, [fp, #-28]
        bl      qdadd
        mov     r3, r0
        str     r3, [fp, #-32]
        ldr     r0, .L3
        ldr     r1, [fp, #-32]
        bl      printf
        mov     r0, r3
        ldmea fp, {fp, sp, pc}
.L4:
        .align          2
.L3:
        .word .LC0
        .size main, .-main
        .ident          "GCC: (GNU) 3.3.1"

Produced assembler file with inline code:


        .file "dsp_qdadd.c"
        .section        .rodata.str1.4,"aMS",%progbits,1
        .align          2
.LC0:
        .ascii          "saime %d\000"
        .text
        .align          2
        .global         main
        .type main, %function
main:
```

```
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        @ link register save eliminated.
        @ lr needed for prologue
        mov   r2, #192
        mov   r0, #113
        qdadd r3,r1,r0
        ldr   r0, .L3
        b     printf
.L4:
        .align      2
.L3:
        .word .LC0
        .size main, .-main
        .align      2
        .global     qdadd
        .type qdadd, %function
qdadd:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        @ link register save eliminated.
        @ lr needed for prologue
        qdadd r3,r2,r1
        mov   r0, r2
        mov   pc, lr
        .size qdadd, .-qdadd
        .ident      "GCC: (GNU) 3.3.1"
```

General Information

If you have any questions regarding the MCS1000 or MosChip Semiconductor, please contact the MosChip Technical Marketing Group at (408) 737-7141 x124 or e-mail info@moschip.com .