Advance Information

AN2469/D
Rev. 1.1, 2/2003

MPC190 PCI Device Driver Design Specification

This document contains information on a new product under development by Freescale. It reflects the current design of the MPC190 device drivers for VxWorks, WindowsNT, and RTLinux. The design of all three drivers will be consistent in future.

This document contains the following topics:

Part I Architecture Overview

The MPC190 is one of the latest Motorola’s security processors which is optimized to process all the algorithms associated with IPSec, IKE, WTLS/WAP and SSL/TLS, including RSA, RSA signature, Diffe-Hellman, elliptic curve, DES, 3DES, SHA-1, MD-4, MD-5 and ARC-4. The MPC190 is designed to operate in a PCI system. The external processors access the MPC190 through its device drivers using system memory for data storage. The MPC190 resides in the PCI address map of the processor, therefore when an application requires cryptographic functions, it creates descriptors for the MPC190, defining the cryptographic function to be performed, and the location of the data. The MPC190 will decode the descriptor and allocate the internal execution unit to do the cryptographic computing. The result is set to the predefine data out buffer and the PCI bus is notified by firing a channel done interrupt. Figure 1-1 shows the physical overview of the MPC190 security processor.

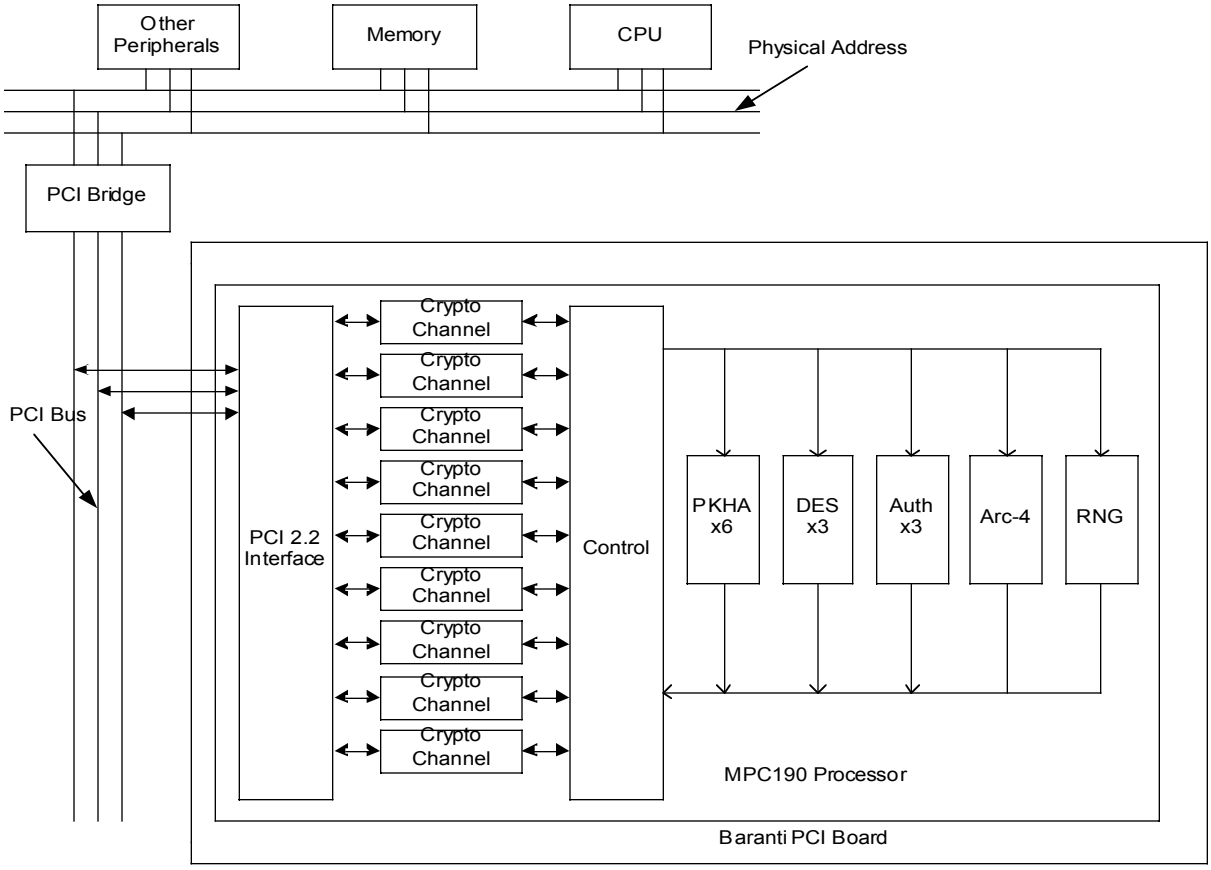

Figure 1-1. Physical Overview

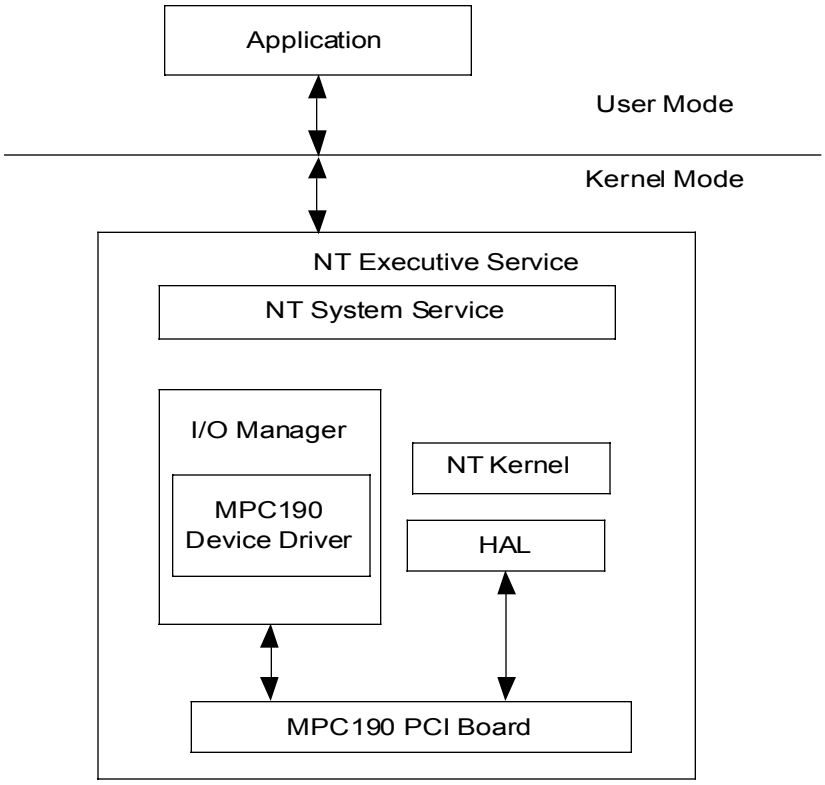

Figure 1-2. Logical Overview for WinNT

Part II Device Driver Structure

General device driver should have these common routines and components:

2.1 Driver Initialize Routine

The device driver will have OS-specific loading and initialization functions. In general, these functions will ensure that the MPC190 is installed and working, that the device driver is properly loaded, and that the requesting task can issue processing requests.

For the Windows NT device driver (Kernel Mode) is a NT service. It can be loaded during the system boot up or after OS system is loaded. During the driver loading phase, the driver initialization routine should find the MPC190, create a NT IoDevice, map the MPC190 physical memory to the NT kernel memory space, allocate global storage and initialize the MPC190 crypto channels and CHAs. The last step is to register the InterruptServiceRoutine to listen to the PCI bus interrupt events.

2.2 IO Request Dispatch Routine

The device driver will have a dispatch function. When a task issues an Io request for processing, the operating system will fire the device driver dispatch routine and pass the Io request context buffer to the dispatch routine as an Irp. The Io request dispatch routine will handle various Irp requests based on the Irp Stack's Major Function and Io control code. If the Irp request is a basic process like IO_Close, IO_Open,

IO_Create, it will directly call these pre-registered functions and normally these functions will complete the Io request. If the Irp request is an Io process request, it will call the Process Request Routine.

2.3 Process Request Routine

The device driver will have a process request function. When the IO Request Dispatch routine sends an Irp to the Process Request, the driver will determine if the MPC190 has enough free resources to process the current request. If it does, the driver will translate the process request into a sequence of one or more Data Packet Descriptors and start the operation. If the MPC190 is busy, the process request will be added to a queue.

2.4 Interrupt Service Routine

The device driver will have an interrupt service routine (ISR), which will be triggered by the PCI INTA interrupt line. Since PCI interrupts are shared, the ISR will first determine if the MPC190 generated an interrupt. If it did, the ISR will clear the interrupt and schedule a separate device driver function to handle the process request completion details. The ISR will be as short and fast as possible.

2.5 Processing Complete Routine

The device driver will have a processing complete function, which is scheduled by the ISR when a processing request is completed, but runs at a lower priority than the ISR. This function will determine which processing requests are complete and notify the corresponding calling tasks. It will then check the processing request queue and based on the available MPC190 resources, initiate one or more processing requests.

WinNT provide DPC mechanism that runs at low level priority to handle these tasks.

2.6 Process Request Queue Routine

The device driver will maintain a processing request queue (protected by a Spin Lock or Mutex, so that the process request function and the processing complete function do not modify the queue at the same time).

The Process Request Queue Routine is normally fired by Processing Complete Routine using ScheduleNext( ) call. It will get one queue_entry and try to find an available channel and CHA resource. If successed, it will move the request from the queue_entry to the ChannelAssignments. Then it removes this queue_entry from the request queue.

2.7 Other Functions

The device driver will have other functions for checking the status of the driver, controlling the driver or MPC190, and setting the block size.

Part III Process Flow Chart

The MPC190 device driver has four basic phases, the initialization phrase, the IO request process phase, interrupt service phase and driver unload phase.

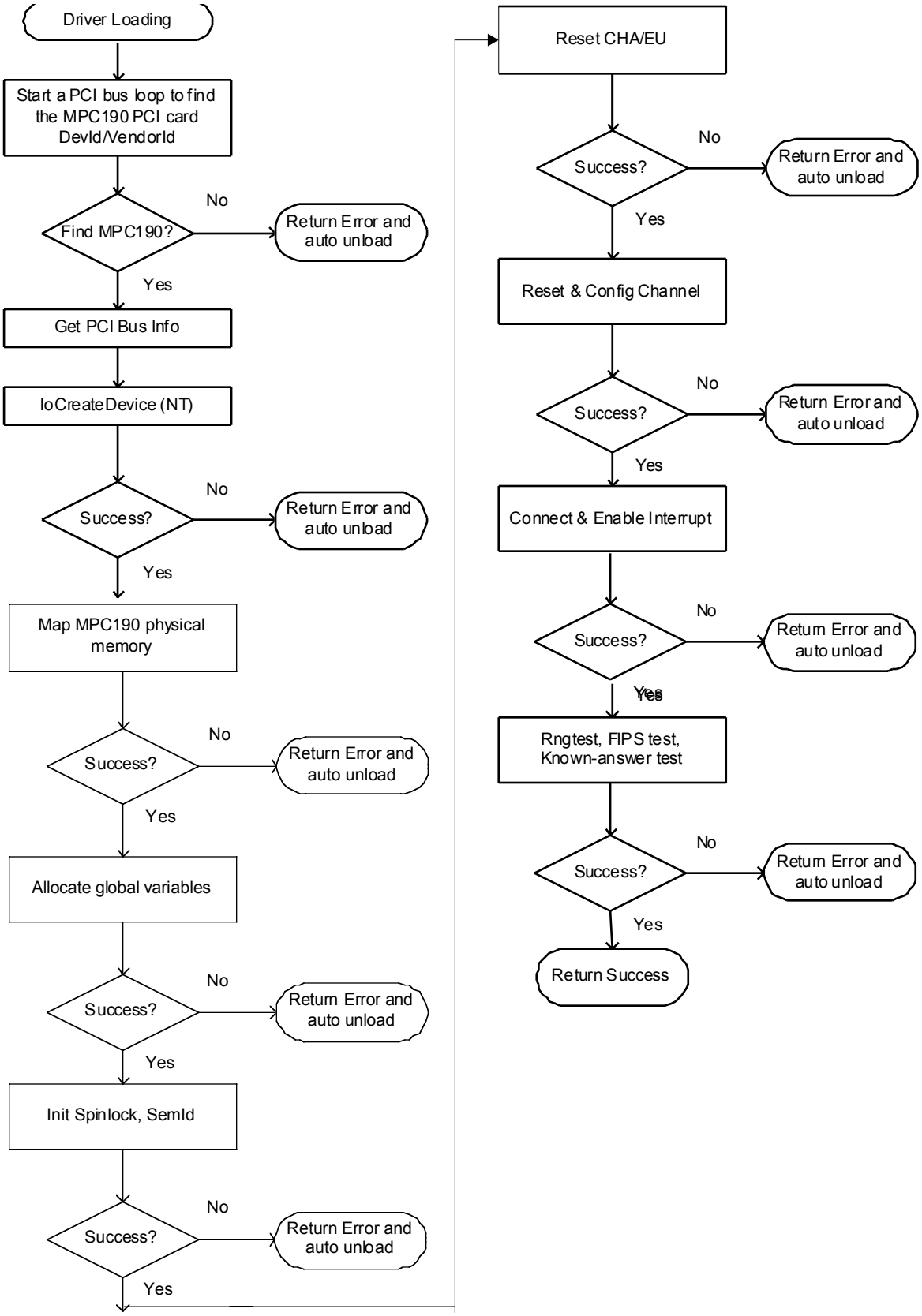

Figure 3-3. Device Driver Initialization.

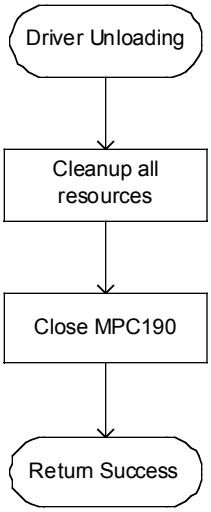

Figure 3-4. Device Driver Unload

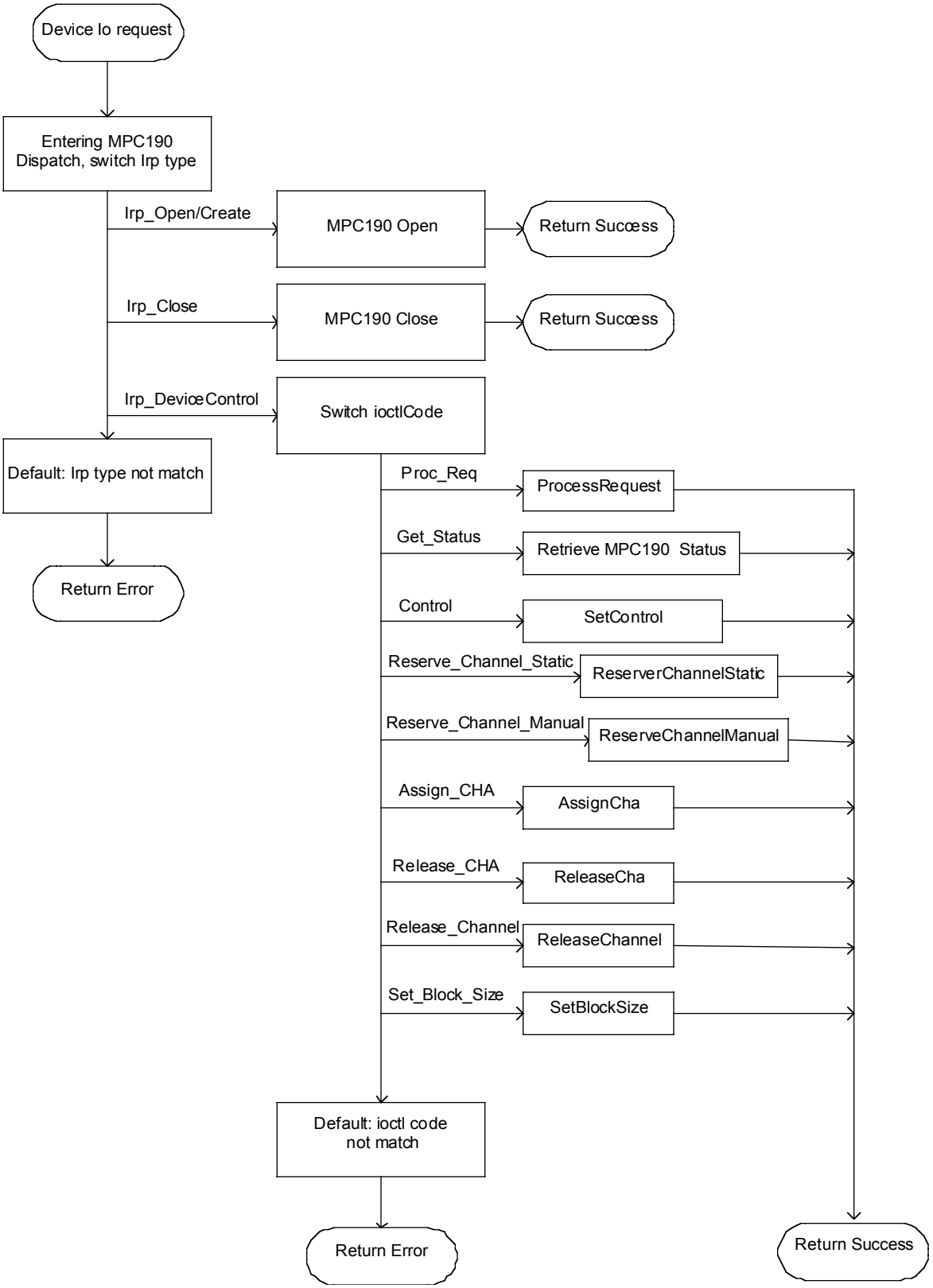

Figure 3-5. IO Request Process

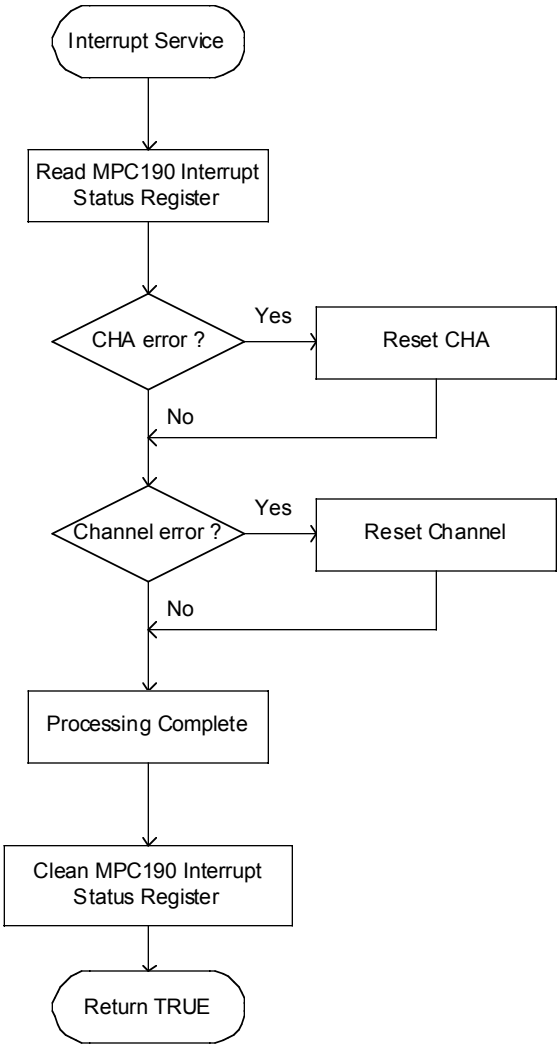

Figure 3-6. Interrupt Service

Part IV Device Driver Interface

The IOCTL device driver function calls have limited capabilities for passing data to and from the device driver (in VxWorks and RTLinux, a single parameter). For cryptographic processing requests, we will use a single parameter to pass in a pointer to a data structure that contains the details of the request. There will be a different process request structure for each type of cryptographic processing supported by MPC190.

The first member of every request structure is an operation ID that can be used by the device driver to determine the format of the rest of the request structure.

All process request structures have a channel member. For process requests that work in either dynamic mode or static mode, the channel can be set to zero to indicate dynamic mode, or to a valid channel number (1 through 9) to indicate static mode. For process requests that only work in static mode, the channel should be set to a valid channel number (1 through 9).

All process request structures have a status member. This value is filled in by the device driver when the interrupt for the operation occurs, and reflects the type of interrupt – done (normal status) or error (error status).

All process request structures have a notify member. This value is used by the device driver to notify the application when its request has been completed.

All process request structures have a next request member. This allows the application to chain multiple process requests together.

The hardware limit of 2048 bytes per data packet descriptor is not exported by the device driver. The application can issue a process request with any length (up to 232 – 1), and the device driver will handle the details of breaking the request up into the proper size chunks.

4.1 Global Variables Definition

The following sections describe channel specific information for channel assignments.

4.1.1 ChannelAssignments

The ChannelAssignments retains all channel specified information, the driver should lock it while modifying. So only one process can modify it at anytime.

typedef struct

```
{
        unsigned char assignment;
        unsigned char isChunked;
        int ownerTaskId;
        void *firstRequest;
        void *currentRequest;
        unsigned long currentOffset;
        void *notify;
        DPD **dpds;
        int dpdCount;

#ifdef WINNT
        PIRP Irp;
        MDL **reqMdl;
        int reqMdlCount;
        MDL **dataMdl;
        int dataMdlCount;
#endif

} CHANNEL_ASSIGNMENT;
CHANNEL_ASSIGNMENT ChannelAssignments[NUM_CHANNELS]
```

4.1.2 ChaAssignments

The ChaAssignments retains the CHA assignment information.

unsigned char ChaAssignments[NUM_CHAS];

4.1.3 ProcessQueueTop and ProcessQueueBottom

These two QUEUE_ENTRY pointer retain the top pointer and the bottom pointer of the processing queue.

QUEUE_ENTRY *ProcessQueueTop;

QUEUE_ENTRY *ProcessQueueBottom;

4.1.4 ChannelAssignLock and BlockSizeLock

The ChannelAssignLock locks the ChannelAssignments modification process while the BlockSizeLock locks the BlockSize modification process.

4.1.5 Other variables

```
int FreeChannels;
int FreeRngas;
int FreeAfhas;
int FreeDesas;
int FreeMdhas;
int FreePkhas;
unsigned long BlockSize; /* The current max block size */
unsigned long PCIBaseAddress; /* The PCI mapping base address */
unsigned long IntStatus[2]; /* Controller interrupt status register, 0x1010 */
unsigned long ChaAssignmentStatus[2];  /*  Controller  EU  assignment  status
register, 0x1028 */
unsigned long ChannelError[NUM_CHANNELS][2];/* Channel Pointer Status Register,
0x2010, 0x3010,...*/
unsigned long ChaError[NUM_CHAS][2]; /* EU/Cha Interrupt Status Register, 10030,
11030,... */
int FIPS_AccessRole;
int FIPS_Connections;
```

4.1.6 Error return codes

```
/* return codes */
#define MPC190_SUCCESS (0)
#define MPC190_MEMORY_ALLOCATION (-1)
#define MPC190_INVALID_CHANNEL (-2)
#define MPC190_INVALID_CHA_TYPE (-3)
#define MPC190_INVALID_OPERATION_ID (-4)
#define MPC190_CHANNEL_NOT_AVAILABLE (-5)
#define MPC190_CHA_NOT_AVAILABLE (-6)
#define MPC190_INVALID_LENGTH (-7)
#define MPC190_OUTPUT_BUFFER_ALIGNMENT (-8)
#define MPC190_RNG_ERROR (-9)

#ifdef WINNT
#define MPC190_PCI_CARD_NOT_FOUNDSTATUS_NO_SUCH_DEVICE
#define MPC190_PCI_MEMORY_ALLOCATE_ERRORSTATUS_INSUFFICIENT_RESOURCES
#else
#define MPC190_PCI_CARD_NOT_FOUND-1000)
#define MPC190_PCI_MEMORY_ALLOCATE_ERROR-1001)
```

```
#endif
#define MPC190_PCI_IO_ERROR-1002)
#define MPC190_PCI_VXWORKS_DRIVER_TABLE_ADD_ERROR    (-1003)
#define MPC190_PCI_INTERRUPT_ALLOCATE_ERROR    (-1004)
```

4.2 Device Driver I/O Interface

This section lists different functions of I/O interface of the device driver.

- Open/Create—This function allows a task to get a device descriptor for future calls to the device driver.
- Close—This function tells the device driver that the user task is finished with its device descriptor.
- IOCTL—This set of functions is the main interface to the device driver. Subfunctions are identified by their IOCTL control code. Subfunctions include:
- Status—Returns the status of the MPC190 card, including crypto channel status, CHA status, and queue length. Third argument in the ioctl() call is a pointer to the MPC190_STATUS structure.
- Control—Allows the caller to modify certain MPC190 features, including Enable/Disable ReserveChannelManual, Enable/Disable ReserveChannelStatic, Enable/Disable Notify, Change the Role Mode (FIPS Crypto Officer, FIPS User, No Control).
- ProcessRequest—Allows the caller to make a request for one or more crypto processing functions. Third argument in ioctl() call is a pointer to a specific request structure.
- ReserveChannelStatic—Statically allocates a channel for use by a single task. Third argument in ioctl() call is an unsigned long specifying the channel number.
- ReserveChannelManual—Allows the caller to reserve a crypto channel for use in manual/debug/target mode. Third argument in ioctl() call is an unsigned long specifying the channel number.
- AssignCHA—Allows the caller to reserve a specific CHA for use by either a static channel or a manual channel. Third argument in ioctl() call is a unsigned long – bottom eight bits are the CHA, next eight bits are the channel number. More than one CHA may be assigned to the same channel by calling this function multiple times. The caller should check the assignment status of the specified CHA (by calling the Status IOCTL function) before trying to assign it.
- ReleaseCHA—Returns a reserved CHA to normal use (dynamic mode) by the device driver. Third argument in ioctl() call is an unsigned long specifying the CHA to release.
- ReleaseChannel—Frees a reserved (static or manual) channel. Third argument in ioctl() call is an unsigned long specifying the channel to release.
- SetBlockSize—Controls the block size that the request data can be broken up to this size (DPD). Third argument in ioctl() call is an unsigned long giving the new block size in bytes. The maximum block size (the hardware upper limit) is 2048 bytes. Default value at driver startup is 2048 bytes.

4.3 Device Driver Internal Functions

This section describe different functions with their associated prototype, platform dependency, input, output and return status.

Table 4-1. Device Driver Internal Functions

Function Name	Platform Depen-dent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
AssignCha	No	To reserve a specific CHA for use by either a static channel or a manual channel.	√	√	√	**channelCha** bottom eight btis is the CHA chaType next eight bits is the channel number (1-9) **currentTaskId** task id	None	MPC190_SUCCESS if success otherwise the error code
			int AssignCha (unsigned long channelCha, int currentTaskId)					
ChaNumToType	No	To translate the the CHA number to the CHA chaType	√	√	√	**cha** CHA Number	None	ChaType if found or MPC190_INVALID_CHA_TYPE if not found.
			int ChaNumToType (int cha)					
CheckChas	No	Check to see if the CHA is avaiable (at least one of this type of CHA is available)	√	√	√	**chaType** CHA Type	None	MPC190_SUCCESS if this type of CHA is available otherwise MPC190_CHA_NOT_AVAILABLE
			int CheckChas (int chaType)					
MPC190 Driver Initialization	Yes	Device driver initialization routine	√	√	√	**DriverObject** PDRIVER_OBJECT (NT only) **RegistryPath** PUNICODE_STRING (NT only)	None	MPC190_SUCCESS if success otherwise the error code
			NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)	int MPC190DriverInit (void)				

Table 4-1. Device Driver Internal Functions (continued)

Function Name	Platform Depen-dent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
InterruptService Routine	Yes	To handle all interrupts generated by the Channel or the CHA (Done or Error) indicated by the Interrupt Status Register . If Error, clean the InterruptStatusRegister by writing InterruptClearRegister. Finally call ProcessingComplete routine.	√	√	√	**Interrupt**	None	TRUE if finished all the steps otherwise FALSE (NT only)
			BOOLEAN MPC190InterruptServiceRoutine (IN PKINTERRUPT Interrupt, IN OUT PVOID Context)	void InterruptServiceRoutine (void)		PKINTERRUPT (NT only)		
						Context		
						PVOID the request Context (NT only)		
OpIdToChaType	No	To translate the crypto operation id to the chaType and later on the routine can check if this type of CHA is available or not.	√	√	√	**OpId**	**chaType**	MPC190_Success if match, otherwise MPC190_INVALID_OPERATION_ID
			int OpIdToChaType (unsigned long OpId, int *chaType)			crypto operation Id	CHA type	
ProcessRequest	Yes	To handle the IO request routine	√	√	√	**req**	**req**	MPC190_Success if success, otherwise MPC190 error code.
			int ProcessRequest (void *req, int callingTaskId, PIRP Irp)	int ProcessRequest (void *req, int callingTaskId)		pointer of the IO request buffer callingTaskId - task Id –	pointer of the IO request buffer	
						Irp		
						PIRP (NT only)		

Table 4-1. Device Driver Internal Functions (continued)

Function Name	Platform Depen-dent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
ReleaseCha	No	To release a specific CHA for use by either a static channel or a manual channel.	√	√	√	**channelCha**	None	MPC190_SUCCESS if success otherwise the error code.
			int ReleaseCha (unsigned long channelCha, int callingTaskId, int locked)			Bottom eight btis are the CHA chaType.the next eight bits are the channel number (1-9)		
						currentTaskId		
						task id		
ReleaseChannel	No	To free a reserved channel (either a static channel or a manual channel.)	√	√	√	**Channel**	None	MPC190_SUCCESS if success otherwise the error code.
			int ReleaseChannel (unsigned long channel, int callingTaskId, int locked)			Channel number (1-9)		
						currentTaskId		
						the task id		
						locked		
						CHANNELS_UNLOCKED or CHANNELS_LOCKED		
ReserveChannelManual	No	To reserve a crypto channel for use in manual/debug mode	√	√	√	**reserve**	Reserve—>channel - the channel number that allocated	MPC190_SUCCESS if success otherwise the error code.
			int ReserveChannelManual (MPC190_RESERVE_MANUAL *reserve, int callingTaskId)			the MPC190_RESERVE_MANUAL structure		
						currentTaskId		
						the task id		
ReserveChannlStatic	No	To allocate a channel for use by a single task.	√	√	√	**channel**	**channel**	MPC190_SUCCESS if success otherwise the error code.
			int ReserveChannelStatic (IN OUT PULONG channel, IN int callingTaskId)			the channel number (1-9)	the channel number that allocated	
						currentTaskId		
						the task id		

Table 4-1. Device Driver Internal Functions (continued)

Function Name	Platform Depen-dent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
SetBlockSize	No	To control how large blocks of data are broken	√	√	√	**newBlockSize**	None	MPC190_SUCCESS if success otherwise the error code.
			int SetBlockSize (unsigned long newBlockSize)			256-2048		
MPC190Open	Yes	To establish an I/O connection between the driver service and the application.	√	√	√	**DeviceObject**	None	MPC190_SUCCESS if success otherwise the error code.
			NTSTATUS MPC190Open (IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)	int MPC190Open (DEV_HDR *pDevHdr, int mode, int flag)	int MPC190Open (struct inode *inode, struct file *filp)	PDEVICE_OBJECT (NT)		
						Irp		
						PIRP (NT)		
MPC190Close	Yes	To close an I/O connection between the driver service and the application.	√	√	√	**DeviceObject**	None	MPC190_SUCCESS if success otherwise the error code.
			NTSTATUS MPC190Close (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)	Prototype (VxWorks) int MPC190Close (int devDesc)	int MPC190Close (int devDesc)	PDEVICE_OBJECT (NT)		
						Irp		
						PIRP (NT)		
						DevDesc		
						device descriptor number		
MPC190Clean-up (NT only)	Yes	To clean up the internal staff before the I/O connection closing.	√			**DeviceObject**	None	MPC190_SUCCESS if success otherwise the error code.
			NTSTATUS MPC190Cleanup(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)			PDEVICE_OBJECT (NT)		
						Irp		
						PIRP (NT)		

Table 4-1. Device Driver Internal Functions (continued)

Function Name	Platform Dependent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
MPC190Unload (NT only)	Yes	To unload itself when the driver is unload.	√			**DeviceObject**	None	MPC190_SUCCESS if success otherwise the error code.
			NTSTATUS MPC190Unload (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)			PDEVICE_OBJECT (NT)		
						Irp		
						PIRP (NT)		
Io Control Dispatch	Yes	To handle the I/O request and dispatch to the different process function based on the ioctl code.	√	√	√	**devDesc**	None	MPC190_SUCCESS if success otherwise the error code.
			NTSTATUS MPC190Dispatch (IN PDEVICE_OBJECT devDesc, IN PIRP Irp)	int Ioctl (int devDesc, int ioctlCode, void *param)	int Ioctl (struct inode *nodePtr, struct file *devDesc, unsigned int ioctlCode, unsigned long param)	PDEVICE_OBJECT (NT)		
						Irp		
						PIRP (NT)		
PCIRead	No	To read numbers of unsigned long from src to dest	√	√	√	**numUlongs**	None	None
			void PCIRead (unsigned long *data, int numUlongs, volatile unsigned long *address)			number of unsigned long to read		
						address		
						the start pointer of the src		
						data		
						the start pointer of the dest.		

Table 4-1. Device Driver Internal Functions (continued)

Function Name	Platform Depen-dent	Purpose	Prototype			Input	Output	Return
			NT	VxWorks	Linux			
PCIWrite	No	T o write numbers of unsigned long from src to dest	√	√	√	**numUlongs**	None	None
			void PCIWrite (unsigned long *data, int numUlongs, volatile unsigned long *address)			number of unsigned long to read		
						address		
						the start pointer of the dest		
						data		
						the start pointer of the src		
ProcessingCom-plete	Yes	To handle the non time-critical process and if the request is done, then complete the I/O.	√	√	√	**Dpc**	None	None
			VOID ProcessingCo mplete (IN PKDPC Dpc, PDEVICE_OB JECT deviceObject, IN PVOID SystemArg1, IN PVOID SystemArg2)	void ProcessingComplete (void)		PKDPC		
						DeviceObject		
						PDEVICE_OBJE CT (NT)		
						SystemArg1		
						PVOID (NT)		
						SystemArg2		
						PVIOD (NT)		
RemoveQueue-Entry	No	To remove the queue from the queue entry chain	√	√	√	**Entry**	None	New queue_ent ry pointer
			QUEUE_ENTRY* RemoveQueueEntry (QUEUE_ENTRY *entry)			pointer of the QUEUE_ENTRY		
RequestToDpd	No	To translate the request structure to DPD chain	√	√	√	**Request**	None	MPC190 _SUCCE SS if success otherwise the error code.
			int RequestToDpd (void *request, int channel)			the request pointer		
						Channel		
						the channel number		
ScheduleNext	No	To process the next request entry in the chain.	√	√	√	None	None	None
			void ScheduleNext (void)					

Part V Design Considerations

The following sections describe the considerations that are needed for designing a multiple platform with MPC190 device.

5.1 Multi Platform Support Considerations

The MPC190 device driver source should support multiple platforms, including WinNT, VxWorks, Linux. For each platform, the MPC190 device driver contains two parts, the platform dependent part and the common part. The common part can be compiled and run under these three platforms. This part only provides the common data processing and computing functionality. The platform dependent part will do OS related tasks, like driver initialization, interrupt service routine and so on.

5.2 PCI Bus interface 32/64 bit words data transfers Issue.

The MPC190 is designed to plug directly into a PCI v2.2 compliant 66 MHz / 64 bit bus. It can be converted to a 33 MHZ / 32 bit 5V bus. Conversion may introduce some problems, e.g. data alignment.

5.3 Big Endian and Little Endian Issue.

WinNT is a little endian system while VxWorks is a big endian system. Big endian introduces word order swap code. A pre-compile flag is recommended to do all of the swap decision code.

5.4 Memory Considerations

[The following discussion applies to WindowsNT and RTLinux, but not VxWorks, which has a unified address space.] Tasks will call the device driver with a pointer to a process request structure allocated from the task's memory space. The device driver runs in kernel space and cannot access the task space directly. Furthermore, when the device driver does gain access to the process request data, it must ensure that the data is page-locked, since a page fault in kernel mode may result in a fatal error. The driver should handle three types of memory space:

5.4.1 For DPDs:

The device driver allocates a fixed amount of kernel memory at startup, and uses this area to buffer the ChannelAssignments[ ]. The DPD chains are inside the ChannelAssignment[ ]. This static kernel memory space is allocated at driver startup, and is deallocated when the driver is unloaded.

5.4.2 For task's input memory space

The device driver can access the task input memory space, but the PCI bus can not access the user memory space. The device driver should map the user's input space to the system. Kernel functions will be used to page-lock the task memory with the User Mode along with an IoReadAccess type. A pointer to the physical memory equivalent will be necessary.

5.4.3 For task's output memory space

The device driver can access the task output memory space, but the PCI bus can not access the user memory space. The device driver should map the user's output space to the system. Kernel functions will be used to page-lock the task memory with the User Mode along with an IoWriteAccess type. A pointer to the physical memory equivalent will be necessary.

5.5 Synchronization Considerations

The MPC190 supports multi channel processing, When the cryptographic execution is done or any error happens, a PCI interrupt will be generated. The device driver's interrupt service routine must figure out which channel or Cha is done and handle the rest of works, e.g. the interrupt clean up, the result data process and IoComplete. From the application point of view, the application has its own user space. The application prepares a request structure, fill in the header field and the rest of length and pointer key pairs. Then it pass the pointer of the request structure to the driver through the DeviceIoControl function. The device driver will accept the request and finish the application's DeviceIoControl call. The application should not expect the result to be ready immediately since the MPC190 device driver handles the output data asynchronously from the DeviceIoControl return. There are two ways to know whether the cryptographic computing is done or not. The first approach is to use a loop to check the request->status field since the device driver will set the status field to 0 when computing is done or to any error code if any error happened. Another approach is to use the notify (callback) mechanism. The application provides a callback entry in the request structure. Then the callback pointer is passed to the device driver along with the request pointer. After the device driver finishes the IoComplete, it will call the callback entry if it is not NULL. The first approach (check the status) is safe but not efficient. The second approach (notify) is highly efficient. The application's callback function should be very atomic and robust. The run time fault or deadly loop of the callback function will damage the device driver and cause the whole system to halt.

5.6 Multi Card Support Considerations

There are two ways to achieve this goal. One is to use a single device driver to control all the MPC190 PCI devices. The driver is responsible to search all the MPC190 devices on the PCI bus and then map each MPC190 PCI physical register to the unique OS memory space and add each MPC190 to the resource table. The device driver is also needed to implement the Io request pool for the dynamic channel request. The advantage of this approach is the simplicity for the application to use this driver for dynamic channel request. The application doesn't need to know which MPC190 card slot number should be used. It only assumes that there will be more logical channels and CHAs available if multi MPC190 devices are used. The disadvantage outweighs the advantage of using only a single driver to control multiple MPC190's. The implementation of the single driver is very complicate. Much more control and code will be needed to handle the multi cards. This will slow down the driver performance. The other drawback is that the single driver can only handle the multi cards Io request sequentially. It also can slow down the overall performance. Further more, if any single MPC190 had strange behavior, e.g. the unexpected interrupt or error, it may block the driver and may affect the rest of all other MPC190 processes. So the second approach (multi drivers for multi cards) is recommended. In this approach, each MPC190 has a unique copy or instance of the device driver. Each driver handles only one MPC190. Different card's Io requests are processed in parallel without interfering with each other. The application is responsible for determining which card to use, in other words, the dynamic channel management should be implemented in the application. The device driver has multiple copies. Each copy contains a unique PCI card number sorted by incremental order. For instance, the MPC190SbDrvNT2.sys will start a loop to find all the MPC190 from the PCI bus and only bind if the second card is found.

5.7 Other Design Considerations/Issues

The following sections describe other design considerations

5.7.1 Thread

The MPC190 device driver is fully interrupt driven. The Io request is handled in the dispatch routine. It processes the request immediately without any delay and returns to the requests without waiting for the chip to finish the cryptographic operation. After the chip finishes the cryptographic operation, it will generate a PCI interrupt. The device driver has the interrupt service routine and some low priority routine to handle the result return process. The application is notified if the notify entry has been passed to the device driver. So from the device driver's point of view, single threading is enough for most of the performance requirements. The device driver itself is thread safe. From the application's point of view, multi threading could increase the speed of Io requests passing into the driver and also increase the speed of data processing after the result is ready. How to design a multi thread application to call the device driver is outside the scope of this document.

5.7.2 Processor

All multi-processor systems need some amount of locking between processors to make sure access to some data structures or hardware is done atomically. The low-level locking code is responsible for serializing such access using spin-locks (a processor will busy-wait while trying to acquire such a lock that has already been locked by the other processor). This operation can tie up resources on some architectures.

The device driver should implement the following two exclusive locks:

ChannelAssignLock – A lock to protect the ChannelAssignments data.

BlockSizeLock – A lock to protect the BlockSize variable.

5.7.3 Interrupt

Most devices generate an interrupt to notify the host computer that they have finished their tasks. The device driver has an interrupt service routine (ISR), which will be triggered by the PCI INTA interrupt line.

Since PCI interrupts are shared, the ISR will first determine if the MPC190 generated the interrupt. If it did, the ISR will clear the interrupt and schedule a separate device driver function to handle the process request completion details. The ISR will be as short and fast as possible.

5.7.3.1 Interrupt Processing For WinNT

- The hardware triggers an interrupt.
- The ISR does the most time-critical processing and clears the interrupt. A DPC is scheduled.
- The DPC routine continues processing and completes the requests (or sets up the hardware and starts the processing of the next portion of a multistage I/O operation).

5.7.3.2 ISR Synchronization

WinNT is intrinsically a multiprocessor system. The ISR services one device, and consequently two processors could be accessing the same device registers or common data area concurrently. This is where the Spin Lock comes in.

5.7.4 Queue entry limitation

Queue entry chain is a dynamic pool of the unprocessed request. The depth of the chain indicates the processing capacity of the driver under channel overloading. The queue entry chain may consume very large amount of system memory since some request may contain a large amount of data. To search and manipulate a large chain of a queue entry may consume a lot of time. The maximize depth of the queue entry is very sensitive to the device driver performance.

5.7.5 Load balance

The MPC190 device driver is designed for withstanding a heavy request process load. For the dynamic request, the device driver can dynamically allocate a free channel and start to process the request without waiting for the other channel to finish a previous operation. The device driver implements the interrupt service routine for the time-critical processing and deferred processing routine for the rest of the I/O process. When the request is completed, the device driver fires the callback function to notify the application that request is done.

5.7.6 Error Handling

The MPC190 device driver should be capable of tolerating most of the error conditions. These error conditions include:

- Incorrect request. The request is empty or any required field is empty. The actual length of the buffer is not equal to the given length.
- OpId is not in the list or the real request structure type is mismatched.
- Invalid notify entry passed in.
- MPC190 channel error
- MPC190 CHA error
- The actual length of the data buffer is over the boundary.
- Spin Lock is deadly locked.
- Memory should be cleaned up and Spin Lock unlocked if any error happened.
- Running of out resources

5.7.7 MPC190Dump

The MPC190 device driver should implement a kernel trace mechanism to dump useful message to the monitoring host. The MPC190Dump supports multiple trace level filters. The precompiled constant MPC190DebugLevel controls which messages level should be dumped. These levels include:

```
#define MPCCONFIG       ((ULONG)0x00000001)
#define MPCUNLOAD        ((ULONG)0x00000002)
#define MPCINITDEV      ((ULONG)0x00000004)
#define MPCIRPPATH      ((ULONG)0x00000008)
#define MPCSTARTER       ((ULONG)0x00000010)
#define MPCPUSHER       ((ULONG)0x00000020
#define MPCERRORS       ((ULONG)0x00000040)
#define MPCTHREAD        ((ULONG)0x00000080)
```

5.7.8 FIPS140-2 Level 1

Motorola's MPC190 Cryptographic Coprocessor System (MCCS) is made up of two distinct components: the MPC190VF cryptographic coprocessor and the PCI driver software used to access the chip's functionality. The MCCS is designed to meet FIPS 140-2 level standard. The following issues should be considered:

- According to the definitions within FIPS 140-2, the MCCS is categorized as a Multi-chip Stand-alone cryptographic module, where the Cryptographic Boundary is defined to be the entire enclosure of the host system.
- The MCCS supports two distinct operator roles: Crypto-Officer role and User role. It does not enforce any identity or role based authentication. The device driver implements this requirement by introducing a flag called Access Role. The application can set the role by calling an I/O function to change the access role type. In other words, the application can choose to play either the Crypto-Officer role or the User role by itself. The device driver only exposes the control services functionality when the application plays as a Crypto-Officer. It only exposes the cryptographic services if the application acts as a User.
- Only one application (or user) at a time can access MCCS even through the host is a multi-tasking and multi-user system. The device driver can enforce this by introducing a user current connection number. If it running under FIPS 140-2 mode, only one user is allowed to connect to the driver.
- If it running under FIPS 140-2 mode, the device driver will only expose the FIPS approved algorithms provided by MCCS: single DES in ECB or CBC, triple DES in ECB or CBC (two keys or three keys), SHA-1 hash and Random Number Generator. The device driver exposes all the algorithms provided by MCCS if under Non FIPS 140-2 mode.
- An application can access MCCS only through the device driver interface. The user level application can not directly access the kernel without the driver interface.
- It is a user/application's responsibility to handle the key storage and key distribution issues in compliance with FIPS 140-2. It is also the responsibility of the application to zero all keys in RAM when it encounters a cryptographic algorithm error and when prior to terminating.
- A power on self test is required for the FIPS 140-2. The test is comprised of the following:
  - Critical function test: Initialize, read and write internal registers in MPC190VF chip.
  - RNG continuous test: The new generated random number should not match the previous. (8 bytest).
  - All cryptographic function test: All of the cryptographic functions by MCCS should be tested.

If any of the self-test fails, a fatal error code is returned to the loader program and the device driver is unloaded. All allocated memory should be zeroed and be free when the loader program exits.

Part VI References

MPC190 Security Co-Processor User's Manual.

Security Policy for the Freescale MPC190 Cryptographic Coprocessor System.

Developing Windows NT Device Drivers, A Programmer's Handbook

Edward N.Dekker Joseph M.Newcomer ISBN 0-201-69590-1

Part VII Acronyms and Abbreviations

This section provides an alphabetical glossary of acronyms and abbreviations used in this document.

- AFHA—ARC-4 Hardware Accelerator.
- ARC-4—Encryption algorithm compatible with the RC-4 algorithm developed by RSA, Inc.
- Auth—Authentication. The CHA or Execution Unit that performs the authentication function is the MDEU, or “Message Digest Execution Unit”.
- CHA—Crypto Hardware Accelerator. This term is synonymous with “Execution Unit” in the MPC190 User’s Manual and other documentation.
- DESA—DES Accelerator.
- DPD—Data Packet Descriptor
- MDHA—Message Digest Hardware Accelerator.
- PKHA—Public Key Hardware Accelerator. This term is synonymous with PKEU in the MPC190 User’s Manual and other documentation.
- RNGA—Random Number Generator Accelerator.

Part VIII Revision History

Table 8-1 summarizes the revision history of this document.

Table 8-1. Revision History

Revision No.	Substantive Change(s)
1	Initial release.
1.1	Added revision history and updated with new template.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

AN2469/D