

DN 6000407

USERS MANUAL
FOR THE ACROAMATICS
MULTI-CHANNEL FRAME SYNCHRONIZER
SOFTWARE

Acroamatics, Inc.
www.acroamatics.com

August 1, 2005

PROPRIETARY NOTICE

Information contained in this document is disclosed in confidence and may not be duplicated in full or in part by any person without prior written approval by Acroamatics, Inc., except as provided by separate contractual agreement. Its sole purpose is to provide the user with adequately detailed documentation in order to install, operate, maintain, and order spare parts for the equipment supplied. The use of this document for any other purpose is expressly prohibited.

ACROAMATICS DOCUMENT HISTORY

The following table indicates major changes made to Users Manual for the Acroamatics Multi-Channel Frame Synchronizer Software , Acroamatics Document Number 6000407, released on August 1, 2005 and contains a record of all revisions made since that date.

DOCUMENT CHANGE HISTORY			
Rev	Date	Action	Name
	08-01-05	Original Issue	TR
A	02-12-07	Describe new routines ReadMcfcsControlSetupEx() and WriteMcfcsControlSetupEx(). Abandon reference to unsupported routines ReadMcfcsControlSetup() and WriteMcfcsControlSetup(). These changes reflect changes to the underlying hardware, in which the MCFS sync registers are now read/write, rather than write only, as they were originally.	TR
B	02-15-07	Added a chapter that is a reference for the data structures used with the MCFS software.	TR
C	06-01-07	Added a chapter describing the Simplified Simulator Setup API. Added to the appendix a description of the data structures used in the Simplified Simulator Setup API.	TR
D	08-26-07	Corrected the document formatting. Added a description in chapter 3 of two new members (MCFS_DECODE and MCFS_INPUT_CLOCK_POLARITY) of the MCFS_CONTROL_REGISTERS data structure.	TR
E	09-20-07	Added a description of the GetMcfcsThreadHandle routine, now included in the McfsData library.	TR
F	08-07-08	Added a description of the LoadSimSetup function in the “Simplified Simulator Setup Interface” chapter.	TR
G	05-20-09	Added description of the McfsGetDataImmediate routine	TR
H	08-31-10	Added description of the McfsRevisionLevel routine.	TR
I	03-24-11	Added descriptions of the Mcfs audio data routines	TR

TABLE OF CONTENTS

CHAPTER 1	1
INTRODUCTION.....	1
CHAPTER 2	1
LOW LEVEL MCFS ACCESS.....	1
mapmcfs.....	1
RMcfsMem16.....	2
WMcfsMem16.....	2
RMcfsDataMem.....	3
WMcfsDataMem.....	3
CHAPTER 3	1
MCFS SETUP INTERFACE	1
McfsResetDevice.....	3
McfsResetChannel.....	4
WriteMcfsControlSetupEx.....	5
ReadMcfsControlSetupEx.....	8
WriteMcfsWordPropertyMem.....	8
ReadMcfsWordPropertyMem.....	11
McfsFrameInitialize.....	12
McfsCommand.....	12
McfsStatus.....	13
IsMcfsChannel.....	14
McfsRevisionLevel.....	14
McfsGetAudioEnabled.....	15
McfsSetAudioRunMode.....	15
McfsGetAudioRunMode.....	15
McfsSetAudioDecoderType.....	16
McfsGetAudioDecoderType.....	16
McfsSetAudioGain.....	17
McfsGetAudioAudioGain.....	17
McfsSetAudioBitRate.....	18
McfsGetAudioBitRate.....	18
McfsSetRawDataType.....	19
McfsGetRawDataType.....	19
CHAPTER 4	1
MCFS DATA INTERFACE	1
McfsDataStart.....	1
McfsGetData.....	3
McfsGetDataImmediate.....	4
McfsDataStop.....	4
GetMcfsThreadHandle.....	5
CHAPTER 5	1
MCFS DATA RECORDING INTERFACE	1
McfsRecordStart.....	1
McfsRecordStop.....	2
CHAPTER 6.....	1
SIMPLIFIED SIMULATOR SETUP INTERFACE.....	1
FormatCreate.....	1
SetCounter.....	2
SetWaveform.....	3
SetFixedWord.....	3
LoadSimSetup.....	4
CHAPTER 7	1
TEST PROGRAMS.....	1

McfsSetup.exe 1
McfsDataTest.exe 1
CHAPTER 8 1
DATA STRUCTURES USED WITH MCFS SOFTWARE 1
 MCFS Setup Interface Data Structures 1
 MCFS Data Interface Data Structures..... 6
 MCFS Data Recorder Interface Data Structures..... 7
 Simplified Simulator Interface Data Structures..... 8

CHAPTER 1 INTRODUCTION

The Acroamatics Model 1626P is a PCI-compatible card that is designed to do basic frame synchronization for up to 4 independent input channels of TTL or RS-422 data. Optional features include an IRIG Time Code Translator and a 64 MHz PCM Format Simulator. Along with the Model 1626 board, Acroamatics delivers software that can be used with programs you write to interface to the MCFS board.

The Acroamatics Multi-Channel Frame Synchronizer (MCFS) Software provides an interface that gives you low-level access to the MCFS registers and memory. There are also interface routines that support MCFS setup, transfer of MCFS telemetry data to your program, and recording MCFS data to your hard disk. These features are implemented in four libraries and one system service program, for use in a Windows operating system environment. Using the library modules allows you to easily write programs to setup and access telemetry data from the MCFS. Full descriptions of the interface and the use of the libraries are presented in the following chapters.

A general-purpose interface giving raw access to the MCFS control registers, word properties memory, and data memory is provided in `cardio.dll`. Link your programs to this interface using `cardio.lib`. The header files `AcroMcf.h` and `A16regs.h` have templates for the MCFS interface and related macros.

The interface for setting up the MCFS is implemented in the library `MCFS.dll`. Use the file `MCFS.lib` to link your programs with this interface. The header file `MCFS.h` provides the templates that comprise this setup interface and other macros that define parameters and return values.

The interface in the library `McfData.dll` lets you read telemetry data from the MCFS board directly into your program's data buffers. You can link your program to this interface using `McfData.lib`. The header file `McfData.h` contains the templates for the routines in this interface as well as some additional macros.

You can control the operations of the system service program called `McfDataRecordingService.exe` with the interface provided by `McfRecord.dll`. You link to this interface by using `McfRecord.lib`. The header file `McfRecord.h` provides the interface templates and other macros. The `McfDataRecordingService.exe` program will record MCFS telemetry data into disk files. The routines in `McfRecord.dll` communicate with this data recording service program through an RPC interface.

CHAPTER 2

LOW LEVEL MCFS ACCESS

A low level interface to all Acroamatics PCI boards can be found in the library called cardio.dll. This library is shipped with all Acroamatics PCI boards, and is documented in Acroamatics document number 6000358, the Application Program Interface Manual. You can refer to this manual for a description of the interface to be used in programming the Time card and Simulator card found on the MCFS board. Interface routines for accessing the MCFS are also included in this library.

There are 5 routines available for MCFS low level access. They are:

```
int mapmcfs(cardno_t unit);

int RMcfsMem16
(
    cardno_t unit,
    unsigned int mcfs_reg,
    regdata_t *ptrBuffer,
    unsigned int count
);

int WMcfsMem16
(
    cardno_t unit,
    unsigned int mcfs_reg,
    regdata_t *ptrBuffer,
    unsigned int count);

int RMcfsDataMem
(
    cardno_t unit,
    unsigned int offset,
    unsigned long *ptrBuffer,
    unsigned int count
);

int WMcfsDataMem
(
    cardno_t unit,
    unsigned int offset,
    unsigned long *ptrBuffer,
    unsigned int count
);
```

mapmcfs

```
int mapmcfs(cardno_t unit);
```

The routine mapmcfs is used to determine if the MCFS card in question is present in the system. The parameter to the function is the unit number. The cardno_t type is define in the header file A16regs.h. The units are numbered starting with 0.

The routine returns a value of -1 if the MCFS board is not present. The return value is 0 if the card is present.

RMcfsMem16

WMcfsMem16

```
int RMcfsMem16
(
    cardno_t unit,
    unsigned int mcfs_reg,
    regdata_t *ptrBuffer,
    unsigned int count
);

int WMcfsMem16
(
    cardno_t unit,
    unsigned int mcfs_reg,
    regdata_t *ptrBuffer,
    unsigned int count
);
```

The routines RMcfsMem16 and WMcfsMem16 are used for accessing the MCFS Word Properties Control Memory. Refer to the Model 1626P hardware manual (Acroamatics document 6000404) for a complete description of the layout and use of this memory. This Word Properties Control Memory is comprised of 16 bit words.

The parameters to these routines are it's unit number (the cardno_t macro is defined in the header file A16regs.h), the word number of the starting memory cell of interest, a pointer to your data buffer into which the MCFS contents will be read, and the number of words to read from the MCFS. These routines return a value of -1 if the memory cannot be accessed, otherwise the return value is 0.

All accesses of the MCFS card made by these routines will be in the 16-bit word memory called the "Word Properties Control Memory". This memory is divided into 8 64K x 16-bits segments. The first four segments are reserved to provide access to the 64 word (16-bit words) PCM Setup and Control Registers. The last 4 64K 16-bit word segments are used for the Word Properties Memory, one segment for each of the 4 MCFS channels.

RMcfsDataMem

WMcfsDataMem

```
int RMcfsDataMem
(
    cardno_t unit,
    unsigned int offset,
    unsigned long *ptrBuffer,
    unsigned int count
);

int WMcfsDataMem
(
    cardno_t unit,
    unsigned int offset,
    unsigned long *ptrBuffer,
    unsigned int count
);
```

The routines RMcfsDataMem and WMcfsDataMem provide access into the 2MB MCFS Output Data Buffer Memory. This memory is divided into 4 segments of 128K 32 bit words. Each segment provides a double-buffer of 64K 32 bit words. One segment for each of the four MCFS channels.

The parameters to this routine are it's unit number (the cardno_t macro is defined in the header file A16regs.h), the word number of the starting memory cell of interest, a pointer to your data buffer into which the MCFS contents will be read, and the number of words to read from the MCFS. These routines return a value of -1 if the memory cannot be accessed, otherwise the return value is 0.

CHAPTER 3

MCFS SETUP INTERFACE

An interface that lets you manage the MCFS Setup Registers and Word Properties is provided. Reading chapter 3 of Acroamatics document 6000404 (MCFS hardware manual) will provide you with the background information that you need to make use of this interface. The routines available in this interface are:

```
MCFS_RETURN_VAL McfsResetDevice(MCFS_DEVICE McfsDevice);
```

```
MCFS_RETURN_VAL McfsResetChannel
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);
```

```
MCFS_RETURN_VAL WriteMcfsControlSetupEx
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_SETUP_DATA ptrMcfsSetup
);
```

```
MCFS_RETURN_VAL ReadMcfsControlSetupEx
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_SETUP_DATA ptrMcfsSetup
);
```

```
MCFS_RETURN_VALUE WriteMcfsWordPropertyMem
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_RANGE ptrRange,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);
```

```
MCFS_RETURN_VALUE ReadMcfsWordPropertyMem
(
    PTR_MCFS_CHANNEL ptrChannel,
    unsigned int WordIndex,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);
```

```
MCFS_API MCFS_RETURN_VAL McfsFrameInitialize
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    unsigned int FrameLength,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);
```

```
MCFS_API MCFS_RETURN_VAL McfsCommand
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_COMMAND McfsCommand
);
```

```
MCFS_API MCFS_RETURN_VAL McfsStatus
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    PTR_MCFS_STATUS ptrMcfsStatus
);

MCFS_API MCFS_RETURN_VAL IsMcfsChannel
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);

MCFS_API unsigned short McfsRevisionLevel
(
    MCFS_DEVICE McfsDevice
);

MCFS_API MCFS_RETURN_VAL McfsGetAudioEnabled
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);

MCFS_API MCFS_RETURN_VAL McfsSetAudioRunMode
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_RUNMODE mode
);

MCFS_API MCFS_RETURN_VAL McfsGetAudioRunMode
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *mode
);

MCFS_API MCFS_RETURN_VAL McfsSetAudioDecoderType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_DECODER decoder
);

MCFS_API MCFS_RETURN_VAL McfsGetAudioDecoderType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *decoder
);

MCFS_API MCFS_RETURN_VAL McfsSetAudioBitRate
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    double bitRate
);
```

```

MCFS_API MCFS_RETURN_VAL McfsGetAudioBitRate
(
    PTR_MCFS_CHANNEL ptrMcfChannel,
    double *bitRate
);

MCFS_API MCFS_RETURN_VAL McfsSetAudioGain
(
    PTR_MCFS_CHANNEL ptrMcfChannel,
    MCFS_AUDIO_GAIN gain
);

MCFS_API MCFS_RETURN_VAL McfsGetAudioGain
(
    PTR_MCFS_CHANNEL ptrMcfChannel,
    int *gain
);

MCFS_API MCFS_RETURN_VAL McfsSetRawDataType
(
    PTR_MCFS_CHANNEL ptrMcfChannel,
    MCFS_AUDIO_RAWDATA_TYPE type
);

MCFS_API MCFS_RETURN_VAL McfsGetRawDataType
(
    PTR_MCFS_CHANNEL ptrMcfChannel,
    int *type
);

```

To use this interface your program must link to the MCFS.lib link library and run with the MCFS.dll dynamic link library. These libraries are shipped with the MCFS SDK.

The return values for all of these routines are defined in an enum called MCFS_RETURN_VALUE in the header file MCFS.h. The various other enums and data structures used as parameters to these routines are also defined as typedefs in the header file MCFS.h.

McfsResetDevice

```
MCFS_RETURN_VAL McfsResetDevice(MCFS_DEVICE McfsDevice);
```

McfsResetDevice is to be used on the MCFS card after it has been powered up, to establish a workable initial state. (Some of the registers on the board power up in an undefined state, and this routine will initialize them properly).

The enum called MCFS_DEVICE is used as the parameter for this routine. Its values are :

- McfsDevice1
- McfsDevice2
- McfsDevice3
- McfsDevice4
- McfsDevice5
- McfsDevice6

- McfsDevice7
- McfsDevice8.

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs.

McfsResetChannel

```
MCFS_RETURN_VAL McfsResetChannel
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);
```

McfsResetChannel is used for resetting a channel on an MCFS card. The parameter to this routine is a pointer to a MCFS_CHANNEL data structure. This structure is defined in MCFS.h as:

```
typedef struct _MCFS_CHANNEL
{
    MCFS_DEVICE McfsDevice;
    MCFS_CHANNEL_NUMBER McfsChannel;
} MCFS_CHANNEL, *PTR_MCFS_CHANNEL;
```

Where MCFS_DEVICE and MCFS_CHANNEL_NUMBER are defined:

```
typedef enum _MCFS_DEVICE
{
    McfsDevice1 = 0,
    McfsDevice2,
    McfsDevice3,
    McfsDevice4,
    McfsDevice5,
    McfsDevice6,
    McfsDevice7,
    McfsDevice8,
    McfsDeviceMax
} MCFS_DEVICE;
```

and

```
typedef enum _MCFS_CHANNEL_NUMBER
{
    McfsChannel1 = 0,
    McfsChannel2,
    McfsChannel3,
    McfsChannel4,
    McfsChannelMax
} MCFS_CHANNEL_NUMBER;
```

These structures and enums are used for telling the various routines in this interface which MCFS card and channel are to be used.

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs.

WriteMcfsControlSetupEx

```
MCFS_RETURN_VAL WriteMcfsControlSetupEx
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_SETUP_DATA ptrMcfsSetup
);
```

The WriteMcfsControlSetupEx routine is used for setting up an MCFS channel control register and its sync pattern and sync mask registers.

The data structures used as parameters to this routine, a pointer to a MCFS_CHANNEL and a pointer to an MCFS_SETUP_DATA data structure are defined in MCFS.h. The MCFS_CHANNEL structure defines the MCFS device and channel numbers, and is illustrated in the section describing the McfsResetChannel routine. The MCFS_SETUP_DATA structure defines the MCFS stream input source, output word formatting flags, and the frame synchronizer strategy. This structure also defines the sync pattern and mask for the stream. The structure is define in MCFS.h as:

```
typedef struct _MCFS_SETUP_DATA
{
    MCFS_CONTROL_REGISTERS McfsControlRegisters;
    unsigned short McfsSyncPattern1;
    unsigned short McfsSyncPattern2;
    unsigned short McfsSyncPattern3;
    unsigned short McfsSyncPattern4;
    unsigned short McfsSyncMask1;
    unsigned short McfsSyncMask2;
    unsigned short McfsSyncMask3;
    unsigned short McfsSyncMask4;
} MCFS_SETUP_DATA, *PTR_MCFS_SETUP_DATA;
```

This structure includes variables that are used to write to the channel's sync pattern and sync mask registers. The MCFS_SETUP_DATA structure contains a MCFS_CONTROL_REGISTERS data structure which is defined in MCFS.h as:

```

typedef struct _MCFS_CONTROL_REGISTERS
{
    MCFS_INPUT_SOURCE McfsInput;
    MCFS_WATCHDOG_TIMER McfsWatchdogTimer;
    MCFS_MESSAGE_WORD_LENGTH McfsMessageWordLength;
    unsigned int McfsVerifyToSearchCount;
    unsigned int McfsVerifyToLockCount;
    unsigned int McfsLockToSearchCount;
    unsigned int McfsErrorToleranceCount;
    unsigned int McfsSyncPatternLength;
    MCFS_VARIABLE_LENGTH_FRAME_POSITION McfsFramePosition;
    MCFS_BIT_SLIP_WINDOW McfsBitSlipWindow;
    MCFS_INPUT_POLARITY McfsInputPolarity;
    MCFS_SYNC_MODE McfsSyncMode;
    MCFS_SYNC_PATTERN_FORMAT McfsSyncPatternFormat;
    MCFS_DECODE McfsDecode;
    MCFS_INPUT_CLOCK_POLARITY McfsInputClockPolarity;
} MCFS_CONTROL_REGISTERS, *PTR_MCFS_CONTROL_REGISTERS;

```

This structure contains unsigned ints and typedef enums that represent settings for the various fields in the MCFS control registers. These registers and their settings are described in Acroamatics document 6000404, chapter 3. The enums contained in the MCFS_CONTROL_REGISTERS structure are defined in MCFS.h as:

```

typedef enum _MCFS_INPUT_SOURCE
{
    McfsSimInput = 0,
    McfsTTLInput,
    Mcfs422Input,
    McfsSerializerInput
} MCFS_INPUT_SOURCE;

typedef enum _MCFS_DOG_TIMER
{
    McfsWatchDogDisable=0,
    McfsWatchDog52Ms=1,
    McfsWatchDog410Us=2,
    McfsWatchDog1_6Us=3
} MCFS_WATCHDOG_TIMER;

typedef enum _MCFS_MESSAGE_WORD_LENGTH
{
    McfsMessageWordlength16 = 0,
    McfsMessageWordlength32
} MCFS_MESSAGE_WORD_LENGTH;

typedef enum _MCFS_VARIABLE_LENGTH_FRAME_POSITION
{
    McfsRandomFramePosition = 0,
    McfsWordSynchronousFramePosition
} MCFS_VARIABLE_LENGTH_FRAME_POSITION;

```

```
typedef enum _MCFS_BIT_SLIP_WINDOW
{
    McfsWindow1Bit = 0,
    McfsWindow3Bit,
    McfsWindow5Bit,
    McfsWindow7Bit
} MCFS_BIT_SLIP_WINDOW;

typedef enum _MCFS_INPUT_POLARITY
{
    McfsPolarityNormal = 0,
    McfsPolarityInvert,
    McfsPolarityAuto
} MCFS_INPUT_POLARITY;

typedef enum _MCFS_SYNC_MODE
{
    McfsSyncModeFixed = 0,
    McfsSyncModeBurst,
    McfsSyncModeAdaptive
} MCFS_SYNC_MODE;

typedef enum _MCFS_SYNC_PATTERN_FORMAT
{
    McfsSyncPatternNormal = 0,
    McfsSyncPatternAltComp,
    McfsSyncPatternComplement
} MCFS_SYNC_PATTERN_FORMAT;

typedef enum _MCFS_DECODE
{
    McfsDecodeNrzl = 0,
    McfsDecodeNrzs
} MCFS_DECODE;

typedef enum _MCFS_INPUT_CLOCK_POLARITY
{
    McfsPolarity0Degree = 0,
    McfsPolarity180Degree
} MCFS_INPUT_CLOCK_POLARITY;
```

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs
- McfsBadStrategyCountValue
- McfsBadStrategyModeValue

ReadMcfsControlSetupEx

```

MCFS_RETURN_VAL ReadMcfsControlSetupEx
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_SETUP_DATA ptrMcfsSetup
);

```

The ReadMcfsControlSetupEx routine is used for reading an MCFS channel control registers.

The data structures used as parameters to this routine, MCFS_CHANNEL and MCFS_SETUP_DATA are defined in MCFS.h and are illustrated in earlier sections of this chapter. MCFS_CHANNEL defines the MCFS device and channel numbers. The MCFS_SETUP_DATA data structure contains elements for a MCFS_CONTROL_REGISTERS structure and elements for the MCFS sync registers (pattern and mask). The MCFS_CONTROL_REGISTERS structure defines the MCFS stream input source, output word formatting flags, and the frame synchronizer strategy.

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs
- McfsSyncRegistersNotReadable

If the return value is McfsSyncRegistersNotReadable it means that your 1626 MCFS board does not have readable sync registers. The other control registers values have been read into your data structure, but your data structure elements used for storing sync pattern and sync mask values have not been modified.

WriteMcfsWordPropertyMem

```

MCFS_RETURN_VALUE WriteMcfsWordPropertyMem
(
    PTR_MCFS_CHANNEL ptrChannel,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_RANGE ptrRange,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);

```

You can use the WriteMcfsWordPropertyMem routine to set the values of a block of words in the MCFS Word Properties Memory.

The data structures used as parameters to this routine, PTR_MCFS_CHANNEL, PTR_MCFS_WORD_PROPERTIES_MEMORY_RANGE and PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP are defined in MCFS.h. MCFS_CHANNEL defines the MCFS device and channel numbers to be used for this operation, and has been illustrated in an earlier section of this chapter. The MCFS_WORD_PROPERTIES_MEMORY_RANGE structure defines the range of the words that you want to modify in the Word Properties Memory. The starting word number and the last word number are specified in this structure, and their values must lie between 0 and 65535 inclusive. This structure is define in MCFS.h as:

```
typedef struct _MCFS_WORD_PROPERTIES_MEMORY_RANGE
{
    unsigned int McfsWpmStart;
    unsigned int McfsWpmEnd;
} MCFS_WORD_PROPERTIES_MEMORY_RANGE,
*PTR_MCFS_WORD_PROPERTIES_MEMORY_RANGE;
```

The MCFS_WORD_PROPERTIES_SETUP defines the specific PCM word characteristics in use. You can specify the number of bits in the word, the orientation, and the justification of the input data. You can enable the word for serialization or suppress the word. You can mark a word as the end of a frame. This structure is defined in MCFS.h as:

```
typedef struct _MCFS_WORD_PROPERTIES_MEMORY_SETUP
{
    MCFS_WPM_WORD_SIZE McfsWordSize;
    MCFS_WPM_WORD_ORIENTATION McfsOrientation;
    MCFS_WPM_DATA_JUSTIFICATION McfsJustification;
    MCFS_WPM_SERIALIZER McfsSerializer;
    MCFS_WPM_SUPPRESS McfsSuppress;
    MCFS_WPM_VARIABLE_LENGTH_FRAME McfsVariableLengthFrame;
    MCFS_WPM_END_OF_FRAME McfsEndOfFrame;
} MCFS_WORD_PROPERTIES_MEMORY_SETUP,
*PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP;
```

The enums used in this structure to define the various word properties are defined in MCFS.h as:

```
typedef enum _MCFS_WPM_WORD_SIZE
{
    McfsWordSize1Bit = 0,
    McfsWordSize2Bits,
    McfsWordSize3Bits,
    McfsWordSize4Bits,
    McfsWordSize5Bits,
    McfsWordSize6Bits,
    McfsWordSize7Bits,
    McfsWordSize8Bits,
    McfsWordSize9Bits,
    McfsWordSize10Bits,
    McfsWordSize11Bits,
    McfsWordSize12Bits,
    McfsWordSize13Bits,
    McfsWordSize14Bits,
    McfsWordSize15Bits,
    McfsWordSize16Bits,
    McfsWordSize17Bits,
    McfsWordSize18Bits,
    McfsWordSize19Bits,
    McfsWordSize20Bits,
    McfsWordSize21Bits,
    McfsWordSize22Bits,
    McfsWordSize23Bits,
    McfsWordSize24Bits,
    McfsWordSize25Bits,
    McfsWordSize26Bits,
    McfsWordSize27Bits,
    McfsWordSize28Bits,
    McfsWordSize29Bits,
    McfsWordSize30Bits,
    McfsWordSize31Bits,
    McfsWordSize32Bits
} MCFS_WPM_WORD_SIZE;

typedef enum _MCFS_WPM_WORD_ORIENTATION
{
    McfsOrientationMsbFirst = 0,
    McfsOrientationLsbFirst
} MCFS_WPM_WORD_ORIENTATION;

typedef enum _MCFS_WPM_DATA_JUSTIFICATION
{
    McfsRightJustified = 0,
    McfsLeftJustified
} MCFS_WPM_DATA_JUSTIFICATION;

typedef enum _MCFS_WPM_SERIALIZER
{
    McfsSerializerDisabled = 0,
    McfsSerailizerEnabled
} MCFS_WPM_SERIALIZER;
```

```

typedef enum _MCFS_WPM_SUPPRESS
{
    McfsSuppressDisabled = 0,
    McfsSuppressEnabled
} MCFS_WPM_SUPPRESS;

typedef enum _MCFS_WPM_VARIABLE_LENGTH_FRAME
{
    McfsVariableLengthFrameDisabled = 0,
    McfsVariableLengthFrameEnabled
} MCFS_WPM_VARIABLE_LENGTH_FRAME;

typedef enum _MCFS_WPM_END_OF_FRAME
{
    McfsEndOfFrameDisabled = 0,
    McfsEndOfFrameEnabled
} MCFS_WPM_END_OF_FRAME;

```

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs
- McfsErrorFrameLength
- McfsWordSizeError

ReadMcfsWordPropertyMem

```

MCFS_RETURN_VALUE ReadMcfsWordPropertyMem
(
    PTR_MCFS_CHANNEL ptrChannel,
    unsigned int WordIndex,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);

```

ReadMcfsWordProperty is used to read the word properties in use for one word in the MCFS Word Properties Memory.

The data structures used as parameters to this routine, MCFS_CHANNEL, and MCFS_WORD_PROPERTIES_MEMORY_SETUP are defined in MCFS.h and have been illustrated in an earlier section of this chapter. MCFS_CHANNEL defines the MCFS device and channel numbers to be used for this operation. The MCFS_WORD_PROPERTIES_SETUP defines the specific PCM word characteristics in use. The WordIndex parameter indicates which word in the Word Properties Memory to read. This value must be between 0 and 65535.

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs
- McfsErrorFrameLength
- McfsWordSizeError

McfsFrameInitialize

```
MCFS_API MCFS_RETURN_VAL McfsFrameInitialize
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    unsigned int FrameLength,
    PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP ptrSetup
);
```

The McfsFrameInitialize routine is used to initialize the word properties memory for a particular MCFS channel. The ptrMcfsChannel parameter is used to declare which MCFS card, and which channel on that card are to be initialized. The FrameLength parameter tells the program how long the frame is to be. The ptrSetup parameter is a pointer to a MCFS_WORD_PROPERTIES_MEMORY_SETUP data structure, which is used to define the desired word properties to this initialization routine. The routine will write “FrameLength” number of words in the channel’s word properties memory, each with the word properties described by the ptrSetup parameter. The “end of frame” bit is also set in the last word written. This routine is useful in situations where contiguous words in a fram will have the same word properties.

The data structures used as parameters to this routine, MCFS_CHANNEL, and MCFS_WORD_PROPERTIES_MEMORY_SETUP are defined in MCFS.h and have been illustrated in an earlier section of this chapter.

The return value for this routine can be:

- McfsSuccess
- McfsDeviceNotFound
- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs
- McfsErrorFrameLength
- McfsErrorFrameLength

McfsCommand

```
MCFS_API MCFS_RETURN_VAL McfsCommand
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_COMMAND McfsCommandWord
);
```

The `McfsCommand` routine is used to start and stop the frame synchronizer on an MCFS channel. The `ptrMcfsChannel` command is used to indicate which MCFS board and channel the command is to be issued to. The `McfsCommandWord` parameter indicates whether the command is run or stop.

The data structures used as parameters to this routine, `MCFS_CHANNEL`, and `MCFS_COMMAND` are defined in `MCFS.h`. `MCFS_CHANNEL` was illustrated in an earlier section of this chapter. The `MCFS_COMMAND` enum is defined as:

```
typedef enum _MCFS_COMMAND
{
    McfsRun = 0,
    McfsStop
} MCFS_COMMAND;
```

The return value for this routine can be:

- `McfsSuccess`
- `McfsDeviceNotFound`
- `McfsBadDeviceNumber`
- `McfsBadChannelNumber`
- `McfsErrorAccessingMcfs`
- `McfsBadCommand`

McfsStatus

```
MCFS_API MCFS_RETURN_VAL McfsStatus
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    PTR_MCFS_STATUS ptrMcfsStatus
);
```

The `McfsStatus` routine is used to read an MCFS channel status register. The `ptrMcfsChannel` parameter tells which MCFS card and which channel on that card are to be used. The elements of the `ptrMcfsStatus` parameter are set according to the values read from the MCFS status register. The `MCFS_CHANNEL` data structure is defined in `MCFS.h` and was illustrated earlier in this chapter.

The data structure `MCFS_STATUS`, used as parameters to this routine, is defined in `MCFS.h` as:

```
typedef struct _MCFS_STATUS
{
    unsigned int ErrorsInSyncPattern;
    int bMoreThan16ErrorsInSyncPattern;
    int bRun;
    int bInputInverted;
    int bLostInputClock;
    int bBitSlipCorrected;
    int bSyncPatternOk;
    MCFS_SYNC_STATUS SyncStatus;
} MCFS_STATUS, *PTR_MCFS_STATUS;
```

The return value for this routine can be:

- `McfsSuccess`
- `McfsDeviceNotFound`

- McfsBadDeviceNumber
- McfsBadChannelNumber
- McfsErrorAccessingMcfs

IsMcfsChannel

```
MCFS_API MCFS_RETURN_VAL IsMcfsChannel
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);
```

The IsMcfsChannel routine can be used to determine if a particular channel is implemented on your 1626 MCFS board. The boards are manufactured with one, two, three or four data channels. This routine allows your program to easily determine which channels are present on a board.

If the channel is present the routine will return a value of McfsSuccess. If the board is not present the routine returns a value of McfsDeviceNotFound. (The MCFS_CHANNEL data structure has elements to specify board number and channel number.) If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber.

McfsRevisionLevel

```
unsigned short McfsRevisionLevel(MCFS_DEVICE McfsDevice);
```

McfsRevisionLevel is used for getting the revision level of an Acroamatics 1626P board.

The enum called MCFS_DEVICE is used as the parameter for this routine. Its values are :

- McfsDevice1
- McfsDevice2
- McfsDevice3
- McfsDevice4
- McfsDevice5
- McfsDevice6
- McfsDevice7
- McfsDevice8.

The return value for this routine is an unsigned short that indicates the board revision level. This function was introduced at board revision level F. A return value less than 5 indicates a revision level that predates revision F. A return value of 5 indicates revision F, a value of 6 will be returned for revision G, and so on.

A return value of 65535 (hexadecimal 0xffff) indicates that the board referenced by the input parameter to this function does not exist in the system.

McfsGetAudioEnabled

```
MCFS_API MCFS_RETURN_VAL McfsGetAudioEnabled
(
    PTR_MCFS_CHANNEL ptrMcfsChannel
);
```

The McfsGetAudioEnabled routine can be used to determine if a 1626 MCFS board is equipped with the audio data capability. You must pass in a pointer to a MCFS_CHANNEL data structure that you use to indicate the device of interest.

If the channel is present and it has audio data capability the routine will return a value of McfsSuccess. If the board is present but has no audio data capability, the routine will return a value of McfsNoAudioFeature. If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber. This routine will return a value of McfsFailed if there is an error reading or writing the MCFS board.

If the board is not present the routine returns a value of McfsDeviceNotFound. (The MCFS_CHANNEL data structure has elements to specify board number and channel number.)

McfsSetAudioRunMode

```
MCFS_API MCFS_RETURN_VAL McfsSetAudioRunMode
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_RUN_MODE runMode
);
```

The McfsSetAudioRunMode routine can be used to set the MCFS audio run mode to AudioRun or AudioStop. These values are provided in an enum in mcfs.h.

If the channel is present and it has audio data capability the routine will set the audio data run state as indicated by the input parameter runMode and return a value of McfsSuccess. If the runMode input parameter is not AudioRun or AudioStop the routine will return a value of McfsBadCommand.

This routine will return a value of McfsFailed if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of McfsNoAudioFeature. If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber.

If the board is not present the routine returns a value of McfsDeviceNotFound. (The MCFS_CHANNEL data structure has elements to specify board number and channel number.)

This routine cannot be used to start or stop the frame synchronizer on the MCFS channel. The McfsCommand routine (described above) must be used for this purpose.

McfsGetAudioRunMode

```
MCFS_API MCFS_RETURN_VAL McfsGetAudioRunMode
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *runMode
);
```

The McfsGetAudioRunMode routine can be used to detect the MCFS audio run mode on a 1626P MCFS board.

If the channel is present and it has audio data capability the routine the integer addressed by the *runMode parameter will be set to indicate the current audio data run state. This value will be either AudioRun or AudioStop, enum values defined in mcfs.h. Then the routine will return a value of McfsSuccess.

This routine will return a value of McfsFailed if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of McfsNoAudioFeature. If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber.

If the board is not present the routine returns a value of McfsDeviceNotFound. (The MCFS_CHANNEL data structure has elements to specify board number and channel number.)

McfsSetAudioDecoderType

```
MCFS_API MCFS_RETURN_VAL McfsSetAudioDecoderType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_DECODER decoderType
);
```

The McfsSetAudioDecoderType routine can be used to set the MCFS audio decoder type to Cvsd or Raw. These values are provided in an enum in mcfs.h.

If the channel is present and it has audio data capability the routine will set the audio decoder type as indicated by the input parameter decoderType and return a value of McfsSuccess. If the input parameter decoderType is a value other than Cvsd or Raw this routine will return a value of McfsBadCommand.

This routine will return a value of McfsFailed if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of McfsNoAudioFeature. If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber.

If the board is not present the routine returns a value of McfsDeviceNotFound. (The MCFS_CHANNEL data structure has elements to specify board number and channel number.)

McfsGetAudioDecoderType

```
MCFS_API MCFS_RETURN_VAL McfsGetAudioDecoderType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *decoderType
);
```

The McfsGetAudioDecoderType routine can be used to detect the MCFS audio decoder type selected on a 1626P MCFS board.

If the channel is present and it has audio data capability the routine the integer addressed by the *decoderType parameter will be set to indicate the current audio decoder type selection. This value will be either Cvsd or Raw, enum values defined in mcfs.h. Then the routine will return a value of McfsSuccess. This routine will return a value of McfsFailed if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of McfsNoAudioFeature. If the channel number is not implemented on the board the routine returns a value of McfsBadChannelNumber.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsSetAudioGain

```
MCFS_API MCFS_RETURN_VAL McfsSetAudioGain
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_GAIN audioGain
);
```

The `McfsSetAudioGain` routine can be used to set the MCFS audio gain to a value between 0 and 5. You can use the enum values `Gain0`, `Gain1`, `Gain2`, `Gain3`, `Gain4` and `Gain5`. These values are provided in an enum in `mcfs.h`.

If the channel is present and it has audio data capability the routine will set the audio gain value as indicated by the input parameter `audioGain` and return a value of `McfsSuccess`. If the input parameter `audioGain` is a value outside of the range of permitted values this routine will return a value of `McfsBadCommand`.

This routine will return a value of `McfsFailed` if there is an error reading or writing the Mcfs board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsGetAudioGain

```
MCFS_API MCFS_RETURN_VAL McfsGetAudioGain
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *audioGain
);
```

The `McfsGetAudioGain` routine can be used to detect the MCFS audio gain selected on a 1626P MCFS board.

If the channel is present and it has audio data capability the routine the integer addressed by the `*audioGain` parameter will be set to indicate the current audio gain selection. This value will be between 0 and 5. Then the routine will return a value of `McfsSuccess`. This routine will return a value of `McfsFailed` if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsSetAudioBitRate

```

MCFS_API MCFS_RETURN_VAL McfsSetAudioBitRate
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    double audioBitRate
);

```

The `McfsSetAudioBitRate` routine can be used to set the `Mcfs` PCM bit rate. The bit rate must be used to set the PCM bit rate when the MCFS audio decoder type is `Cvsd`, and the use of this function is valid only if the current MCFS decoder type selection is `Cvsd`. If you use this function when the decoder type selection is `Raw` the routine will return an error value of `McfsAudioCommandNotValidForCurrentDecoderType`.

If the channel is present and it has audio data capability, and the current decoder type is `Cvsd` the routine will set the PCM bit rate as indicated by the input parameter `audioBitRate` (expressing bits per second) and return a value of `McfsSuccess`. If the input parameter `audioBitRate` is a value larger than 64Mb/s, this routine will return a value of `McfsAudioBitRateError`. (The maximum PCM bit rate supported by the MCFS is 64Mb/s.)

The PCM rate is used, along with values stored in the MCFS word properties memory, to calculate a decoder clock divider and the sample clock divider. The sample clock divider expresses the audio sample rate as a divisor of the decoder clock, which is in turn related to the PCM bit rate. The calculated audio sample rate must be between 8Kb/s and 64Kb/s. If the rate is outside this range, the routine will return an error value of `McfsAudioSampleRateError`. If this error occurs you may have declared the wrong PCM bit rate. Or it may be that your PCM data frame is described incorrectly. Possible frame description errors are: frame length is wrong; the words in the frame that carry the audio data are declared incorrectly. These frame characteristics are stored in the MCFS channel word properties memory, and this memory must be set before using the `McfsSetAudioBitRate` routine.

This routine will also inspect the frame word property values to verify that each word selected as an audio data source meets the requirement that audio data must be MSB-first orientation. If this requirement is not met this routine will return a warning code of `McfsAudioInvalidWordPropertySetting`.

This routine will return a value of `McfsFailed` if there is an error reading or writing the `Mcfs` board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsGetAudioBitRate

```

MCFS_API MCFS_RETURN_VAL McfsGetAudioBitRate
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    double *audioBitRate
);

```

The `McfsGetAudioBitRate` routine can be used to read the MCFS PCM bit rate selected on a 1626P MCFS board. This routine may be used only when the MCFS audio decoder type is `Cvsd`. If you use this routine when the `Mcfs` audio decoder type is not `Cvsd` it will return an error value of `McfsAudioCommandNotValidForCurrentDecoderType`.

If the channel is present and it has audio data capability, and the audio decoder type is `Cvsd`, the double addressed by the `*audioBitRate` parameter will be set to indicate the current PCM bit rate selection. Then

the routine will return a value of `McfsSuccess`. This routine will return a value of `McfsFailed` if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsSetRawDataType

```
MCFS_API MCFS_RETURN_VAL McfsSetRawDataType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    MCFS_AUDIO_RAWDATA_TYPE dataType
);
```

The `McfsSetRawDataType` routine can be used to set the MCFS audio raw data type to `Obn` (offset binary) or `Tcm` (two's complement). You can use the enum values provided in `mcfs.h`.

If the channel is present and it has audio data capability the routine will set the audio raw data type as indicated by the input parameter `dataType` and return a value of `McfsSuccess`. If the input parameter `dataType` is a value outside of the range of permitted values this routine will return a value of `McfsBadCommand`.

Using this function is valid only when the MCFS decoder type selection is `Raw`. If you attempt to use this function when the decoder selection is `Cvsd` an error code of `McfsAudioCommandNotValidForCurrentDecoderType` will be returned.

This routine will return a value of `McfsFailed` if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

McfsGetRawDataType

```
MCFS_API MCFS_RETURN_VAL McfsGetRawDataType
(
    PTR_MCFS_CHANNEL ptrMcfsChannel,
    int *dataType
);
```

The `McfsGetRawDataType` routine can be used to detect the MCFS audio raw data type selected on a 1626P MCFS board.

If the channel is present and it has audio data capability the routine the integer addressed by the `*dataType` parameter will be set to indicate the current raw data type selection. This will be a value of `Obn` or `Tcm`. (These enum values are defined in `mcfs.h`.) Then the routine will return a value of `McfsSuccess`.

Using this function is valid only when the MCFS decoder type selection is `Raw`. If you attempt to use this function when the decoder selection is `Cvsd` an error code of `McfsAudioCommandNotValidForCurrentDecoderType` will be returned.

This routine will return a value of `McfsFailed` if there is an error reading or writing the MCFS board.

If the board is present but has no audio data capability, the routine will return a value of `McfsNoAudioFeature`. If the channel number is not implemented on the board the routine returns a value of `McfsBadChannelNumber`.

If the board is not present the routine returns a value of `McfsDeviceNotFound`. (The `MCFS_CHANNEL` data structure has elements to specify board number and channel number.)

CHAPTER 4

MCFS DATA INTERFACE

After you have setup your Model 1626P Frame Synchronizer and are using it to process your data, you can use the Mcfs Data interface to read telemetry data from MCFS channels into your program. To use this interface your program must link to the McfsData.lib library, and run with McfsData.dll. These libraries are shipped with the MCFS SDK. The routines available in this interface are:

```

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfDataStart(PTR_MCFS_START_DATA pStart);

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfDataStop(PTR_MCFS_STOP_DATA pStop);

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfGetData(PTR_MCFS_GET_DATA pGetData, unsigned int
*Length);

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfGetDataImmediate(PTR_MCFS_GET_DATA pGetData, unsigned
int *Length);

MCFSDATA_API HANDLE
GetMcfThreadHandle(unsigned int uiDeviceNumber);

```

The return values for all of these routines are defined in an enum called MCFS_RETURN_VALUE in the header file McfsData.h. The various other enums and data structures used as parameters to these routines are also defined as typedefs in the header file McfsData.h.

The interface works by calling the McfsDataStart routine when you want to start reading data from the card. You then call McfsDataGet to read frames or records of data. When your program is finished reading data from the MCFS card, call the McfsDataStop to close the data channel.

Depending upon the data rate on the Mcfs output and depending also on your computer system characteristics you may need to adjust the priority class at which your program runs. For example if your data rate is very fast or if your system includes processes that run too long or too often at a very high priority class, you may find that using the NORMAL_PRIORITY_CLASS for your program is not adequate. The McfsData library starts a separate thread for each model 1626P board in your system. While gathering data from the 1626P these threads must respond in a timely fashion to interrupts from the model 1626P. Excessive delays caused by other processes running at higher priority may not be tolerable and can lead to loss of data through overflow. The McfsData library leaves it to you to select the priority class at which your program runs. Furthermore, you may also want to adjust the thread priority for any McfsData thread. The GetMcfThreadHandle routine is provided to give you a handle to any of the library threads that respond to 1626P interrupts. You can use this handle to adjust the thread priority of any of these McfsData threads. By setting your process priority class and modifying thread priority you can set any scheduling priority for a McfsData thread.

McfDataStart

```

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfDataStart(PTR_MCFS_START_DATA pStart);

```

The first step in getting MCFS data into you program is to establish a connection to the MCFS Data interface using the McfsDataStart routine. You pass this routine a pointer to a MCFS_START_DATA data structure. This data structure is defined as:

```
typedef struct _MCFS_START_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned int RecordLength;
    int bControlRunState;
    int bFrameMode;
    int bEnableTime;
    int bEnableStatus;
    HANDLE hDataEvent;
} MCFS_START_DATA, *PTR_MCFS_START_DATA;
```

The DeviceNumber field is used to select the MCFS card number, while the ChannelNumber field is used to select the channel on the card. Each card has up to four channels, depending on which configuration you buy. The hDataEvent can be used to specify a wait event that will be signaled when the next buffer of data is ready. To use this feature, pass in a valid event handle to the McfsDataStart routine, then wait for that event to be signaled before calling the McfsGetData routine. If you do not want to use this feature, pass NULL as the value hDataEvent.

This interface can be run in frame mode, or in record mode. In frame mode the data becomes available at the end of each telemetry frame. In record mode you specify the number of 16 bit words in your record, and the data becomes available when a record has been filled. Set the bFrame element in the MCFS_START_DATA structure to 1 when you want to run in frame mode. Set it to zero when you want to run in record mode. When you get data in record mode, use the RecordLength element of the input data structure to tell the interface how long you want the records to be. The maximum number of 16 bit words in a record is 131072.

Acroamatics recommends that you use record mode when collecting your data. The advantage in using record mode rather than frame mode is that record mode allows the use of larger record lengths, which makes the McfsGetData routine operate more efficiently with higher data rates. In frame mode there is one DMA transfer and one interrupt per frame of data. In record mode your record length can be many times larger than the length of a single data frame, thereby using fewer CPU cycles servicing DMA operations and interrupts. Use of frame mode is suitable for very slow data rates, or if you do not need to receive every data frame.

The MCFS can stamp each frame of data with a header comprised of time and/or status. Set the bEnableTime element of the MCFS_START_DATA structure to 1 if you want the time stamp, set it to zero if you do not. Set the bEnableStatus element of the input data structure to 1 if you want each frame stamped with the status word, otherwise set bEnableStatus to zero.

If you like, the MCFS Data interface will put the data channel into run when you call McfsDataStart and stop the channel when you call McfsDataStop. To select this behavior set the bControlRunState element of the MCFS_START_DATA structure to one. If you prefer to control the channel run state yourself some other way, set this element of the input data structure to zero.

If the routine is able to start the data on the MCFS channel it will return the value McfsDataSuccess. If the routine is unable to start the data the return value will indicate the reason the start operation failed.

The return value for this routine can be:

- McfsDataSuccess
- McfsDataInvalidDevice
- McfsDataInvalidChannel
- McfsDataDeviceNotFound
- McfsDataChannelBusy

McfsGetData

```

MCFSDATA_API MCFSDATA_RETURN_VAL
McfsGetData
(
    PTR_MCFS_GET_DATA pGetData,
    unsigned int *Length
);

```

After the MCFS channel has been opened with a call to `McfsDataStart` you can read telemetry data from the MCFS card into your program memory by calling the `McfsGetData` routine. The parameters you pass in to this routine are a pointer to a `MCFS_GET_DATA` data structure and a pointer to an unsigned integer that this routine will use to store the number of 16 bit words of data that were copied into your data buffer. The `MCFS_GET_DATA` structure is defined as:

```

typedef struct _MCFS_GET_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned short *DataBuffer;
    unsigned int DataBufferSize;
} MCFS_GET_DATA, *PTR_MCFS_GET_DATA;

```

The `DataBuffer` element of the `MCFS_GET_DATA` structure is a pointer to the program memory where you want the MCFS telemetry data copied. The `DataBufferSize` element of this structure tells the routine the size of your buffer, measure in 16 bit words. The `DeviceNumber` and `ChannelNumber` fields are used to indicate which device and channel are to be used for data.

If there is a frame or record (depending on the mode you choose in the `McfsDataStart` call) of data ready this routine will copy the data into your buffer and return a value of `McfsDataSuccess`. The `Length` parameter will be used to store the number of 16 bit words transferred. If your memory buffer is smaller than the frame or record you will not get the entire data set. The maximum size for a frame or record is 131072 16 bit words. If at this time the MCFS hardware indicates that data has been lost due to data overflow the return value will be `McfsDataOverflow`. This will happen if your program is unable to read the data at the rate of the output of the MCFS channel. You can continue to use this routine in this situation, but it means that you are not getting every frame or record.

If a record or frame has not been completed on the MCFS channel when you make the call to the `McfsGetData` routine no data will be copied into your buffer, and the routine will return a value of `McfsDataNotReady`.

If the interface is being used in frame mode and the `McfsGetData` routine returns an error code that indicates a DMA error has occurred (`McfsDataFailed`, `McfsDmaError`, or `McfsDataOverflowDmaError`), the use of the DMA will be abandoned on future `McfsData` routine calls in favor of the use of a memory copy routine to transfer the frame data from the 1626P device.

If you make a call to the `McfsGetData` routine without having successfully made a call to the `McfsDataStart` routine, `McfsGetData` will copy no data into your buffer and will return a value of `McfsDataNotStarted`.

McfsGetDataImmediate

```

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfsGetDataImmediate
(
    PTR_MCFS_GET_DATA pGetData,
    unsigned int *Length
);

```

The McfsGetDataImmediate routine is useful in situations when there is an incomplete record or buffer of data that you want to retrieve. The parameters you pass in to this routine are a pointer to a MCFS_GET_DATA data structure and a pointer to an unsigned integer that this routine will use to store the number of 16 bit words of data that were copied into your data buffer. The MCFS_GET_DATA structure is defined as:

```

typedef struct _MCFS_GET_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned short *DataBuffer;
    unsigned int DataBufferSize;
} MCFS_GET_DATA, *PTR_MCFS_GET_DATA;

```

The DataBuffer element of the MCFS_GET_DATA structure is a pointer to the program memory where you want the MCFS telemetry data copied. The DataBufferSize element of this structure tells the routine the size of your buffer, measure in 16 bit words. The DeviceNumber and ChannelNumber fields are used to indicate which device and channel are to be used for data.

If there is a frame or record (depending on the mode you choose in the McfsDataStart call) of data ready this routine will copy the data into you buffer and return a value of McfsDataSuccess. The Length parameter will be used to store the number of 16 bit words transferred. If your memory buffer is smaller than the frame or record you will not get the entire data set. The maximum size for a frame or record is 131072 16 bit words. If at this time the MCFS hardware indicates that data has been lost due to data overflow the return value will be McfsDataOverflow. This will happen if your program is unable to read the data at the rate of the output of the MCFS channel. You can continue to use this routine in this situation, but it means that you are not getting every frame or record.

If a record or frame has not been completed on the MCFS channel when you make the call to the McfsGetDataImmediate routine the entire buffer of record or frame data will be copied from the Model 1626P data buffer into your program memory. Since the data in the 1626P buffer represents an incomplete record or frame, it is likely that the buffer contains fresh data followed by stale data. The length of the full record will be returned in the Length parameter variable. This Length value returned does not indicate the number of 16-bit words of fresh data, but rather the number of 16-bit words of fresh and stale data.

If you make a call to the McfsGetDataImmediate routine without having successfully made a call to the McfsDataStart routine, McfsGetData will copy no data into you buffer and will return a value of McfsDataNotStarted.

McfsDataStop

```

MCFSDATA_API MCFS_DATA_RETURN_VAL
McfsDataStop(PTR_MCFS_STOP_DATA pStop);

```

When your program is finished reading data from the MCFS card using the `McfsDataGet` routine it must call the `McfsDataStop` routine to disconnect from the channel. You pass this routine a pointer to a `MCFS_STOP_DATA` data structure. This structure is defined as:

```
typedef struct _MCFS_STOP_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    int bControlRunState;
} MCFS_STOP_DATA, *PTR_MCFS_STOP_DATA;
```

The `DeviceNumber` field is used to select the MCFS card number, while the `ChannelNumber` field is used to select the channel on the card. Each card has up to four channels, depending on which configuration you buy.

Use the `bControlRunState` element of the `MCFS_STOP_DATA` structure to make the routine control the MCFS run state. If `bControlState` is set to one, the routine will place the MCFS channel into idle state if the channel was idle when the `McfsDataStart` routine was called. If `bControlState` is zero, `McfsDataStop` will not modify the MCFS channel run state.

The return value for this routine can be:

- `McfsDataSuccess`
- `McfsDataInvalidDevice`
- `McfsDataInvalidChannel`
- `McfsDataDeviceNotFound`
- `McfsDataIllegalStopCommand`

The `McfsDataIllegalStopCommand` return value is given if the MCFS device and channel parameters are not the same device and channel number that was used in the `McfsDataStart` routine.

GetMcfsThreadHandle

```
MCFSDATA_API HANDLE
GetMcfsThreadHandle(unsigned int uiDeviceNumber);
```

The `GetMcfsThreadHandle` routine is used get a handle to any of the `McfsData` threads that respond to interrupts from a model 1626P board. The input parameter `uiDeviceNumber` should range in value from 0 to 7. The routine will return a handle to the interrupt thread or will return 0 if there is no active interrupt thread for the device. You can modify the thread scheduling priority by using this handle to set the thread priority.

CHAPTER 5

MCFS DATA RECORDING INTERFACE

The Model 1626P Multi Channel Frame Synchronizer software includes a program called McfsRecordServer.exe. This program runs as a Windows system service program that starts up automatically when you start up your system. This program can take the telemetry data output from a 1626P card and write into a disk file on your computer, or the data can be sent to a multicast or point-to-point UDP address on the local area network. The program supports an RPC interface that allows other programs to direct its data recording activities.

An interface is provided that makes it easy for your program to use this data recording service program to record MCFS telemetry data. The routines available in this interface are:

```
MCFSRECORD_API MCFS_RECORD_RETURN_VAL
McfRecordStart(PTR_MCFS_START_RECORD pStart);

MCFSRECORD_API MCFS_RECORD_RETURN_VAL
McfRecordStop(PTR_MCFS_STOP_RECORD pStop);
```

The return values returned by these routines are typedef enums that are defined in the header file McfsRecord.h. To use this interface your program must link to the McfsRecord.lib library, and run with the McfsRecord.dll dynamic link library.

McfRecordStart

```
MCFSRECORD_API MCFS_RECORD_RETURN_VAL
McfRecordStart(PTR_MCFS_START_RECORD pStart);
```

Use the routine McfsRecordStart to direct the system service program named McfsRecordServer.exe to record MCFS telemetry data into a disk file. The parameter to this routine is a pointer to a MCFS_START_RECORD data structure, which is defined by:

```
typedef struct _MCFS_START_RECORD
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned int RecordLength;
    unsigned char *FileName;
    int bControlRunState;
    int bFrameMode;
    int bEnableTime;
    int bEnableStatus;
} MCFS_START_RECORD, *PTR_MCFS_START_RECORD;
```

The DeviceNumber field is used to select the MCFS card number, while the ChannelNumber field is used to select the channel on the card. Each card has up to four channels, depending on which configuration you buy.

The FileName member of the MCFS_START_RECORD tells the record server what the name of the data file will be or selects multicast or point-to-point UDP address and port number. The syntax for this socket address specification is IP Address:portnumber. For example the string "224.187.225.255:7765" can be used to select the multicast UDP socket at port number 7765 and IP address 224.187.225.255 for data distribution, while the string "192.168.55.136:7765" can be used to select the point-to-point UDP socket at port 7765 and IP address 192.168.55.136 for data distribution. If the IP address you choose lies within the range of 224.0.0.0 through 239.255.255.255 a multicast UDP socket will be used. Otherwise a point-to-point UDP socket will be used.

The record operation interface can be run in frame mode, or in record mode. In frame mode the data becomes available at the end of each telemetry frame. In record mode you specify the number of 16 bit words in your record. Set the bFrame element in the MCFS_START_RECORD structure to 1 when you want to run in frame mode. Set it to zero when you want to run in record mode. When you use record mode, use the RecordLength element of the input data structure to tell the program how many 16 bit words you want in the records. The maximum number of 16 bit words in a record is 131072.

The MCFS can stamp each frame of data with a header comprised of time and/or status. Set the bEnableTime element of the MCFS_START_DATA structure to 1 if you want the time stamp, set it to zero if you do not. Set the bEnableStatus element of the input data structure to 1 if you want each frame stamped with the status word, otherwise set bEnableStatus to zero.

If you like, the McfsRecordServer.exe program will put the data channel into run when you start recording data and stop the channel when you stop recording data. To select this behavior set the bControlRunState element of the MCFS_START_DATA structure to one. If you prefer to control the channel run state yourself some other way, set this element of the input data structure to zero.

If the data recording operation is started the routine's return value will be McfsRecordSuccess. All other return values from this routine will indicate that a data recording operation was not started, and will indicate why the data recording could not be started.

The return value for this routine can be:

- McfsRecordSuccess
- McfsRecordFailed
- McfsRecordChannelBusy
- McfsRecordInvalidDevice
- McfsRecordInvalidRecordLength
- McfsRecordInvalidChannel
- McfsRecordDeviceNotFound
- McfsRecordFileOpenError
- McfsRecordMemoryAllocationError
- McfsRecordDriverBufferError
- McfsRecordDmaOpenError
- McfsRecordDataError
- McfsRecordServerUnavailable

McfsRecordStop

```
MCFSRECORD_API MCFS_RECORD_RETURN_VAL  
McfsRecordStop(PTR_MCFS_STOP_RECORD pStop);
```

When you have recorded all the MCFS telemetry data that you want, your program must call McfsRecordStop to complete the data recording. The parameter to this routine is a pointer to a MCFS_STOP_RECORD data structure, defined as:

```
typedef struct _MCFS_STOP_RECORD
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    int bControlRunState;
    int DataWriteErrorCode;
} MCFS_STOP_RECORD, *PTR_MCFS_STOP_RECORD;
```

Once again the DeviceNumber field is used to select the MCFS card number, while the ChannelNumber field is used to select the channel on the card. Each card has up to four channels, depending on which configuration you buy.

If you like, the McfsRecordServer.exe program will put the data channel into run when you start recording data and stop the channel when you stop recording data. To select this behavior set the bControlRunState element of the MCFS_STOP_DATA structure to one. If you prefer to control the channel run state yourself some other way, set this element of the input data structure to zero.

The return value for this routine can be:

- McfsRecordSuccess
- McfsRecordInvalidDevice
- McfsRecordInvalidChannel
- McfsRecordDeviceNotFound
- McfsRecordIllegalStopCommand
- McfsRecordDataError
- McfsRecordServerUnavailable

When the return value from the McfsRecordStop routine is McfsRecordDataError the DataWriteErrorCode element of the MCFS_STOP_RECORD data structure that you used as a parameter will contain information about the exact nature of the error in the data recording.

The DataWriteErrorCode member of the data structure is used by the record server program to give you details about any errors that might have occurred during the writing of the telemetry data to the disk file. This is a bit encoded word with the following error bits defined in McfsDataRecordErrors.h :

```
#define MCFS_RECORD_HARDWARE_OVERFLOW_ERROR 1
#define MCFS_RECORD_BUFFER_OVERFLOW_ERROR 2
#define MCFS_RECORD_FILE_WRITE_ERROR 4
#define MCFS_RECORD_DMA_TRANSFER_ERROR 8
#define MCFS_RECORD_THREAD_TERMINATE_ERROR 16
#define MCFS_RECORD_FIFO_OVERFLOW_ERROR 32
#define MCFS_RECORD_INSTRUCTION_MEMORY_ERROR 64
#define MCFS_RECORD_WORD_PROCESSING_ERROR 128
```

The MCFS_RECORD_HARDWARE_OVERFLOW_ERROR bit is set if the hardware buffer on the MCFS card had an overflow error during the data recording error. This can happen when the data recording interrupt service routine cannot process the MCFS data fast enough to catch every data frame or record generated by the MCFS card. This means that the data that is recorded does not include all the data that was generated by the MCFS card.

The MCFS_RECORD_BUFFER_OVERFLOW_ERROR bit will be set if there was an overflow of a software buffer during the data recording. This can happen if the carry away rate of the data to the hard disk is not fast enough for the data rate of the output from the MCFS card. This error means that some of the data was lost during the recording.

The MCFS_RECORD_FILE_WRITE_ERROR bit will be set if there was a disk write error during the data recording. This error means that some of the data recorded may not be recoverable.

The MCFS_RECORD_DMA_TRANSFER_ERROR bit will be set if the low level DMA transfer routine used to move telemetry data from the MCFS card into the record server buffer pool returned an error condition during the data recording. This error may mean that some of the data recorded is erroneous.

The MCFS_RECORD_THREAD_TERMINATE_ERROR bit is set if the McfsRecordStop routine is unable to terminate any thread that was in use during the recording. This error does not usually involve the loss of data, but may mean the server is in an unstable state.

The MCFS_RECORD_FIFO_OVERFLOW_ERROR will be set if the FIFO used to fill the hardware data buffers on the MCFS card overflowed during the data recording. This error means that some data was lost during the recording.

The MCFS_RECORD_INSTRUCTION_MEMORY_ERROR and the MCFS_RECORD_WORD_PROCESSING_ERROR bits indicate a hardware error in the MCFS frame synchronizer during the data recording. These errors are described in the MCfs hardware manual, Acroamatics document number 6000404.

CHAPTER 6

SIMPLIFIED SIMULATOR SETUP INTERFACE

This chapter outlines a simple interface to the programmable simulator. (The 64 MHz PCM Format Simulator is an option on the 1626P MCFS board.) This interface provides a subset of the standard Acroamatics Simulator setup interface, one which is compatible with the features of the 1626P MCFS frame synchronizer.

This interface is provided in a dynamic link library for use in a Windows operating system environment. Use the file named SimpleSimApi.lib for linking your programs. Templates for the interface functions and other definitions can be found in the header files SimpleSimApi.h and SimpleSimTypes.h.

The interface is comprised of five functions. One function, called FormatCreate, lets you setup the structure and other characteristics of the frame you want to generate. These setup features are loaded immediately to the simulator board.

Three functions are used to define the data resources that will be used to generate your simulated data. These functions are used to assign selected data words to a resource. The resources are: Counters (2), Waveform (2), and Fixed Words. These setup values are not written immediately to the simulator board, but are accumulated in program memory. When you have finished defining your data resource setup, use the LoadSimSetup function to transfer this setup information to the simulator board. It is an error to attempt to associate a data source resource with a format word without having first created the format using the FormatCreate() function.

All SimpleSimApi functions return a bool data type, a value of true indicates that the operation was completed successfully. The first parameter to all SimpleSimApi functions is an integer that indicates the card number of the simulator being programmed. The value must be between 0 and 7.

FormatCreate

```
SIMPLESIM_API bool FormatCreate
(
    int cardnumber,
    int wordsize,
    int numberofwords,
    long syncpattern,
    float bitrate,
    enum Codetype,
    enum Orientation
);
```

Use the “FormatCreate” function to define the simulation frame format. The “cardnumber” parameter indicates which simulator you are programming. The “wordsize” parameter sets the word size of each word in the frame. The word size is the number of bits per word in every word in the frame. This value must lie between 4 and 16.

The “numberofwords” parameter defines the number of words in each frame. When the format is defined, all words will be set to “random” data, except for the words allocated for the sync pattern. The number of words in each frame must lie between 4 and 32768.

The “syncpattern” parameter defines the sync pattern generated for your simulated data frame. The sync pattern will determine how many words at the end of the frame are occupied. If the word size is 10 bits, for example, and the sync pattern is given as 0x370, only the last word in the frame will be used for the pattern, even though the hexadecimal value could be interpreted as having 12 bits (3 hex digits). The pattern will be interpreted as having a length that is a multiple of the word size. When building the simulation, as many fixed data words, “wordsize” in length, as necessary will be used to generate the

sync pattern. If the word length is 12 bits, and the pattern is 0xFAF320, the last two words in the frame will be used as syllables of the sync word. If the word length is 8 bits, with the same sync pattern, the last three words of the frame will be used.

The “bitrate” parameter is the data rate of the stream, in cycles per second. Look in the header file “SimpleSimTypes.h” or in the appendix of this document for definitions of the enums you can use to select values for the “Codetype” and “Orientation” parameters.

The simulator setup features that are programmed with this function are written immediately to the simulator board.

SetCounter

```
SIMPLESIM_API bool SetCounter
(
    int cardnumber,
    int counternumber,
    int wordnumber,
    int wordinterval,
    unsigned short preset,
    unsigned short limit
);
```

The PCM Format Simulator option on the 1626P MCFS board has two counter-word resources that can be used to generate data in your simulated telemetry data stream. Use the “SetCounter” function to define these counters.

The “cardnumber” parameter indicates which simulator you are programming. The “counternumber” parameter indicates which counter you are setting up, and its value must be 0 or 1.

The “wordnumber” and “wordinterval” parameters are used to indicate where in the frame the counter-word data is to be placed. Word numbering starts with 1. If the counter data will occupy only one word in the frame, use 0 as the value for the “wordinterval” parameter. If the counter data will occupy every word in the frame, use 1 as the value for the “wordinterval” parameter; use 2 as the value for the “wordinterval” value if the counter data will occupy alternate frame words, and so on. It is illegal to assign a resource to a word allocated as part (or all) of the sync pattern. If a word interval is specified, the sync word is automatically excluded.

The “preset” parameter will indicate the starting value of the counter, the “limit” parameter will indicate the ending value of the counter. The counter will be set to increment at Mainframe Recycle.

No more than one format word may be assigned to a counter or a waveform, except when supercommutation is defined by specifying a word interval. This means that the software will keep track of the words assigned to those resources, and when a new assignment is made, the previously assigned words are set to output random data.

The simulator setup features that are programmed with this function are not written immediately to the simulator board. Rather they are accumulated in program memory. When all of your simulator data source definitions have been completed you must use the LoadSimSetup() function to load your setups to the simulator board.

SetWaveform

```
SIMPLESIM_API bool SetWaveform
(
    int cardnumber,
    int waveformnumber,
    int wordnumber,
    int wordinterval,
    float rate,
    enum Waveform,
    enum Datatype
);
```

The PCM Format Simulator option on the 1626P MCFS board has two waveform-word resources that can be used to generate data in your simulated telemetry data stream. Use the “SetWaveform” function to define these resources.

The “cardnumber” parameter indicates which simulator you are programming. The “waveformnumber” parameter indicates which waveform you are setting up, and its value must be 0 or 1. The waveform data resources can generate ramp, sine, triangle, or square waveforms.

The “wordnumber” and “wordinterval” parameters are used to indicate where in the frame the waveform data is to be placed. Word numbering starts with 1. If the waveform data will occupy only one word in the frame, use 0 as the value for the “wordinterval” parameter. If the waveform data will occupy every word in the frame, use 1 as the value for the “wordinterval” parameter; use 2 as the value for the “wordinterval” value if the waveform data will occupy alternate frame words, and so on. It is illegal to assign a resource to a word allocated as part (or all) of the sync pattern. If a word interval is specified, the sync word is automatically excluded.

The “rate” parameter specifies the frequency (in cycles per second) of the waveform function. The rate range is .000003 through 12203. If the rate parameter is set to 0, the waveform data will increment at every reference, making the wave period a function of the reference rate.

The “Waveform” parameter will indicate which wave form you will generate. You can select from square, triangle, ramp, and sine wave data. Look in the header file “SimpleSimTypes.h” or in the appendix of this document for definitions of the enums you can use to select values for the “Waveform” and “Datatype” parameters.

No more than one format word may be assigned to a counter or a waveform, except when supercommutation is defined by specifying a word interval. This means that the software will keep track of the words assigned to those resources, and when a new assignment is made, the previously assigned words are set to output random data.

The simulator setup features that are programmed with this function are not written immediately to the simulator board. Rather they are accumulated in program memory. When all of your simulator data source definitions have been completed you must use the LoadSimSetup() function to load your setups to the simulator board.

SetFixedWord

```
SIMPLESIM_API bool SetFixedWord
(
    int cardnumber,
    long value,
    int wordnumber,
    int wordinterval
);
```

The PCM Format Simulator option on the 1626P MCFS board has 1024 fixed-word resources that can be used to generate data in your simulated telemetry data stream. Use the “SetFixedWord” function to define these resources.

The “cardnumber” parameter indicates which simulator you are programming

The “wordnumber” and “wordinterval” parameters are used to indicate where in the frame the fixed data word is to be placed. Word numbering starts with 1. If the fixed word data will occupy only one word in the frame, use 0 as the value for the “wordinterval” parameter. If the fixed word data will occupy every word in the frame, use 1 as the value for the “wordinterval” parameter; use 2 as the value for the “wordinterval” value if the fixed word data will occupy alternate frame words, and so on. It is illegal to assign a resource to a word allocated as part (or all) of the sync pattern. If a word interval is specified, the sync word is automatically excluded.

The “value” parameter is used for setting the value of the fixed word data. You specify the data value, and one of the simulator’s 1024 fixed data word resources will be programmed for that value. You can program as many fixed data words as you have available, until you run out.

Setting a fixed data word does nothing to previously programmed words, unless they were assigned to a counter or waveform before (in which case the previously assigned resource is marked as “unused”). If a fixed data word is assigned to any word that had previously been assigned to a counter or waveform by means of word number plus word interval, all words in the series will be reassigned. For example, if a counter had been assigned to word 8, with a word interval of 8, a fixed word assignment to word 16 would not only put the fixed word value in that word, but would put random data in words 8, 24, *etc.*, and the counter previously assigned would be marked as unused.

One or more of the fixed data words will have been used for the sync pattern. If the software keeps track of the simulator’s fixed words, and what they have been set for, a given simulator data word could be assigned to more than one format word. If the user assigned a value of 0x1234 to word 10, for example, then later assigned the same value to word 18, the same simulator data word could be used to drive both format words. Fixed data words are output as programmed (*i.e.*, not affected by the format orientation).

The simulator setup features that are programmed with this function are not written immediately to the simulator board. Rather they are accumulated in program memory. When all of your simulator data source definitions have been completed you must use the LoadSimSetup() function to load your setups to the simulator board.

LoadSimSetup

```
SIMPLESIM_API bool LoadSimSetup
(
    int cardnumber
);
```

The “cardnumber” parameter indicates which simulator you are programming

Use this function after you have completed all of your data source definitions with the SetCounter(), SetWaveform() and SetFixedWord() functions. The LoadSimSetup() function will transfer the data source definitions that these other functions have stored in program memory to the simulator board.

CHAPTER 7

TEST PROGRAMS

The Acroamatics SDK for the Model 1626P includes two programs that use the interfaces described in the first chapters of this document. These test programs are provided so that you can compare the results of your programs with these.

McfsSetup.exe

McfsSetup.exe is a windows program that uses the interfaces in Low Level MCFS Access (chapter 2) and MCFS Setup INterface (chapter 3) to setup the MCFS to process your telemetry data. Controls are provided to let you reset the card (which should be the first step you take in setting up the card), to setup the control registers, to setup the word properties memory, and to run and stop a channel. The channel lock status (idle/search/verify/lock) is displayed.

McfsDataTest.exe

McfsDataTest.exe is a WIN32 console program that uses the MCFS Data Interface to transfer telemetry data from an MCFS channel into its data buffer. The program accepts command line arguments that let you control the details of the data transfers. You can specify the card number, channel number, frame or record mode, and enable or disable headers with command line arguments. If you give the common line argument “-?” the program will give a message that lists the command line syntax and gives a description of the commands.

The program then calls the McfsDataStart routine to establish a connection to the output of an MCFS channel. Then the McfsGetData routine is called repeatedly to fill the program’s data buffer with data from the channel. The program will keep track of the status returned on each call to McfsGetData and display a summary when you finish transferring data. Typing any key except “q” will make the program display the contents of the current buffer. Up to 80 words of the frame or record will be displayed.

Striking the “q” key causes the program to stop calling McfsGetData and display a summary of the values returned by McfsGetData. The program will then call McfsDataStart to disconnect from the MCFS channel before it exits.

CHAPTER 8

DATA STRUCTURES USED WITH MCFS SOFTWARE

The following is a description of the data structures used with the MCFS software. The descriptions are presented as they appear in C programming language header files.

MCFS Setup Interface Data Structures

```

typedef enum _MCFS_RETURN_VAL
{
    McfsSuccess = 1,
    McfsFailed,
    McfsDeviceNotFound,
    McfsBadDeviceNumber,
    McfsBadChannelNumber,
    McfsBadStrategyCountValue,
    McfsBadStrategyModeValue,
    McfsErrorAccessingMcfs,
    McfsErrorFrameLength,
    McfsWordSizeError,
    McfsBadCommand,
    McfsSyncRegistersNotReadable,
    McfsLastError
} MCFS_RETURN_VAL;

typedef enum _MCFS_DEVICE
{
    McfsDevice1 = 0,
    McfsDevice2,
    McfsDevice3,
    McfsDevice4,
    McfsDevice5,
    McfsDevice6,
    McfsDevice7,
    McfsDevice8,
    McfsDeviceMax
} MCFS_DEVICE;

typedef enum _MCFS_COMMAND
{
    McfsRun = 0,
    McfsStop
} MCFS_COMMAND;

typedef enum _MCFS_CHANNEL_NUMBER
{
    McfsChannel1 = 0,
    McfsChannel2,
    McfsChannel3,
    McfsChannel4,
    McfsChannelMax
} MCFS_CHANNEL_NUMBER;

```

```
typedef enum _MCFS_INPUT_SOURCE
{
    McfsSimInput = 0,
    McfsTTLInput,
    Mcfs422Input,
    McfsSerializerInput
} MCFS_INPUT_SOURCE;

typedef enum _MCFS_WATCHDOG_TIMER
{
    McfsWatchDogDisable=0,
    McfsWatchDog52Ms=1,
    McfsWatchDog410Us=2,
    McfsWatchDog1_6Us=3
} MCFS_WATCHDOG_TIMER;

typedef enum _MCFS_MESSAGE_WORD_LENGTH
{
    McfsMessageWordlength32 = 0,
    McfsMessageWordlength16
} MCFS_MESSAGE_WORD_LENGTH;

typedef enum _MCFS_DECODE
{
    McfsDecodeNrzl = 0,
    McfsDecodeNrzs
} MCFS_DECODE;

typedef enum _MCFS_INPUT_CLOCK_POLARITY
{
    McfsPolarity0Degree = 0,
    McfsPolarity180Degree
} MCFS_INPUT_CLOCK_POLARITY;

typedef enum _MCFS_VARIABLE_LENGTH_FRAME_POSITION
{
    McfsRandomFramePosition = 0,
    McfsWordSynchronousFramePosition
} MCFS_VARIABLE_LENGTH_FRAME_POSITION;

typedef enum _MCFS_BIT_SLIP_WINDOW
{
    McfsWindow1Bit = 0,
    McfsWindow3Bit,
    McfsWindow5Bit,
    McfsWindow7Bit
} MCFS_BIT_SLIP_WINDOW;

typedef enum _MCFS_INPUT_POLARITY
{
    McfsPolarityNormal = 0,
    McfsPolarityInvert,
    McfsPolarityAuto,
    McfsPolarityAutoInvert
} MCFS_INPUT_POLARITY;
```

```
typedef enum _MCFS_SYNC_MODE
{
    McfsSyncModeFixed = 0,
    McfsSyncModeBurst,
    McfsSyncModeAdaptive
} MCFS_SYNC_MODE;

typedef enum _MCFS_SYNC_PATTERN_FORMAT
{
    McfsSyncPatternNormal = 0,
    McfsSyncPatternAltComp,
    McfsSyncPatternComplement
} MCFS_SYNC_PATTERN_FORMAT;

typedef enum _MCFS_WPM_WORD_SIZE
{
    McfsWordSize1Bit = 0,
    McfsWordSize2Bits,
    McfsWordSize3Bits,
    McfsWordSize4Bits,
    McfsWordSize5Bits,
    McfsWordSize6Bits,
    McfsWordSize7Bits,
    McfsWordSize8Bits,
    McfsWordSize9Bits,
    McfsWordSize10Bits,
    McfsWordSize11Bits,
    McfsWordSize12Bits,
    McfsWordSize13Bits,
    McfsWordSize14Bits,
    McfsWordSize15Bits,
    McfsWordSize16Bits,
    McfsWordSize17Bits,
    McfsWordSize18Bits,
    McfsWordSize19Bits,
    McfsWordSize20Bits,
    McfsWordSize21Bits,
    McfsWordSize22Bits,
    McfsWordSize23Bits,
    McfsWordSize24Bits,
    McfsWordSize25Bits,
    McfsWordSize26Bits,
    McfsWordSize27Bits,
    McfsWordSize28Bits,
    McfsWordSize29Bits,
    McfsWordSize30Bits,
    McfsWordSize31Bits,
    McfsWordSize32Bits
} MCFS_WPM_WORD_SIZE;

typedef enum _MCFS_WPM_WORD_ORIENTATION
{
    McfsOrientationMsbFirst = 0,
    McfsOrientationLsbFirst
} MCFS_WPM_WORD_ORIENTATION;
```

```
typedef enum _MCFS_WPM_DATA_JUSTIFICATION
{
    McfsRightJustified = 0,
    McfsLeftJustified
} MCFS_WPM_DATA_JUSTIFICATION;

typedef enum _MCFS_WPM_SERIALIZER
{
    McfsSerializerDisabled = 0,
    McfsSerailizerEnabled
} MCFS_WPM_SERIALIZER;

typedef enum _MCFS_WPM_SUPPRESS
{
    McfsSuppressDisabled = 0,
    McfsSuppressEnabled
} MCFS_WPM_SUPPRESS;

typedef enum _MCFS_WPM_VARIABLE_LENGTH_FRAME
{
    McfsVariableLengthFrameDisabled = 0,
    McfsVariableLengthFrameEnabled
} MCFS_WPM_VARIABLE_LENGTH_FRAME;

typedef enum _MCFS_WPM_END_OF_FRAME
{
    McfsEndOfFrameDisabled = 0,
    McfsEndOfFrameEnabled
} MCFS_WPM_END_OF_FRAME;

typedef enum _MCFS_SYNC_STATUS
{
    Search = 0,
    Verify,
    Lock
} MCFS_SYNC_STATUS;

typedef struct _MCFS_CHANNEL
{
    MCFS_DEVICE McfsDevice;
    MCFS_CHANNEL_NUMBER McfsChannel;
} MCFS_CHANNEL, *PTR_MCFS_CHANNEL;
```

```

typedef struct _MCFS_CONTROL_REGISTERS
{
    MCFS_INPUT_SOURCE McfsInput;
    MCFS_WATCHDOG_TIMER McfsWatchdogTimer;
    MCFS_MESSAGE_WORD_LENGTH McfsMessageWordLength;

    unsigned int McfsVerifyToSearchCount;
    unsigned int McfsVerifyToLockCount;
    unsigned int McfsLockToSearchCount;
    unsigned int McfsErrorToleranceCount;
    unsigned int McfsSyncPatternLength;

    MCFS_VARIABLE_LENGTH_FRAME_POSITION McfsFramePosition;
    MCFS_BIT_SLIP_WINDOW McfsBitSlipWindow;
    MCFS_INPUT_POLARITY McfsInputPolarity;
    MCFS_SYNC_MODE McfsSyncMode;
    MCFS_SYNC_PATTERN_FORMAT McfsSyncPatternFormat;
    MCFS_DECODE McfsDecode;
    MCFS_INPUT_CLOCK_POLARITY McfsInputClockPolarity;
} MCFS_CONTROL_REGISTERS, *PTR_MCFS_CONTROL_REGISTERS;

typedef struct _MCFS_SETUP_DATA
{
    MCFS_CONTROL_REGISTERS McfsControlRegisters;

    unsigned short McfsSyncPattern1;
    unsigned short McfsSyncPattern2;
    unsigned short McfsSyncPattern3;
    unsigned short McfsSyncPattern4;

    unsigned short McfsSyncMask1;
    unsigned short McfsSyncMask2;
    unsigned short McfsSyncMask3;
    unsigned short McfsSyncMask4;
} MCFS_SETUP_DATA, *PTR_MCFS_SETUP_DATA;

typedef struct _MCFS_WORD_PROPERTIES_MEMORY_RANGE
{
    unsigned int McfsWpmStart;
    unsigned int McfsWpmEnd;
} MCFS_WORD_PROPERTIES_MEMORY_RANGE,
*PTR_MCFS_WORD_PROPERTIES_MEMORY_RANGE;

typedef struct _MCFS_WORD_PROPERTIES_MEMORY_SETUP
{
    MCFS_WPM_WORD_SIZE McfsWordSize;
    MCFS_WPM_WORD_ORIENTATION McfsOrientation;
    MCFS_WPM_DATA_JUSTIFICATION McfsJustification;
    MCFS_WPM_SERIALIZER McfsSerializer;
    MCFS_WPM_SUPPRESS McfsSuppress;
    MCFS_WPM_AUDIO_SOURCE McfsAudioSource;
    MCFS_WPM_VARIABLE_LENGTH_FRAME McfsVariableLengthFrame;
    MCFS_WPM_END_OF_FRAME McfsEndOfFrame;
} MCFS_WORD_PROPERTIES_MEMORY_SETUP,
*PTR_MCFS_WORD_PROPERTIES_MEMORY_SETUP;

```

```

typedef struct _MCFS_STATUS
{
    unsigned int ErrorsInSyncPattern;
    int bMoreThan16ErrorsInSyncPattern;
    int bRun;
    int bInputInverted;
    int bLostInputClock;
    int bBitSlipCorrected;
    int bSyncPatternOk;
    MCFS_SYNC_STATUS SyncStatus;
} MCFS_STATUS, *PTR_MCFS_STATUS;

```

MCFS Data Interface Data Structures

```

typedef enum _MCFS_DATA_RETURN_VAL
{
    McfsDataSuccess      = 1,
    McfsDataFailed,
    McfsDataChannelBusy,
    McfsDataInvalidDevice,
    McfsDataInvalidRecordLength,
    McfsDataInvalidChannel,
    McfsDataDeviceNotFound,
    McfsDataIllegalStopCommand,
    McfsDataNotStarted,
    McfsDataNotReady,
    McfsDataOverflow,
    McfsDmaError,
    McfsDataOverflowDmaError,
    McfsDataLastError
} MCFS_DATA_RETURN_VAL;

typedef struct _MCFS_START_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned int RecordLength;
    int bControlRunState;
    int bFrameMode;
    int bEnableTime;
    int bEnableStatus;
    HANDLE hDataEvent;
} MCFS_START_DATA, *PTR_MCFS_START_DATA;

typedef struct _MCFS_STOP_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    int bControlRunState;
} MCFS_STOP_DATA, *PTR_MCFS_STOP_DATA;

```

```
typedef struct _MCFS_GET_DATA
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned short *DataBuffer;
    unsigned int DataBufferSize;
} MCFS_GET_DATA, *PTR_MCFS_GET_DATA;
```

MCFS Data Recorder Interface Data Structures

```
typedef enum _MCFS_RECORD_RETURN_VAL
{
    McfsRecordSuccess = 1,
    McfsRecordFailed,
    McfsRecordChannelBusy,
    McfsRecordInvalidDevice,
    McfsRecordInvalidRecordLength,
    McfsRecordInvalidChannel,
    McfsRecordDeviceNotFound,
    McfsRecordFileOpenError,
    McfsRecordWriteThreadError,
    McfsRecordIllegalStopCommand,
    McfsRecordMemoryAllocationError,
    McfsRecordDriverBufferError,
    McfsRecordDmaOpenError,
    McfsRecordThreadTerminateError,
    McfsRecordDataError,
    McfsRecordServerUnavailable,
    McfsRecordLastError
} MCFS_RECORD_RETURN_VAL;
```

```
typedef struct _MCFS_START_RECORD
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    unsigned int RecordLength;
    unsigned char *FileName;
    int bControlRunState;
    int bFrameMode;
    int bEnableTime;
    int bEnableStatus;
} MCFS_START_RECORD, *PTR_MCFS_START_RECORD;
```

```
typedef struct _MCFS_STOP_RECORD
{
    unsigned int DeviceNumber;
    unsigned int ChannelNumber;
    int bControlRunState;
    int DataWriteErrorCode;
} MCFS_STOP_RECORD, *PTR_MCFS_STOP_RECORD;
```

Simplified Simulator Interface Data Structures

```
enum Orientation
{
    SsMSBWOrientation=1,
    SsLSBWOrientation=0
};
```

```
Enum Codetype
{
    SsNRZL=0,
    SsNRZM=1,
    SsNRZS=2,
    SsDBIPM=3,
    SsRNRZ11=4,
    SsRNRZ15=5,
    SsRNRZ17=6,
    SsRNRZ23=7,
    SsDBIPS=8,
    SsBIPL=9,
    SsBIPM=10,
    SsBIPS=11,
    SsDMM=12,
    SsDMS=13,
    SsDMMM=14,
    SsDMSS=15
};
```

```
enum Waveform
{
    SsRamp=0,
    SsSine=1,
    SsSquareWave=2,
    SsTriangle=3
};
```

```
enum Datatype
{
    SsOnesComp=0,
    SsTwosComp=1,
    SsSignMagnitude=2,
    SsOffsetBinary=3
};
```