FMSTERSCIDRVUG
Rev. 3.0, 08/2016

FreeMASTER Serial Communication Driver

Table of Contents

Chapter 4 PLATFORM-SPECIFIC TOPICS

List of Tables

Chapter 1 INTRODUCTION

FreeMASTER is a PC-based application serving as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units. This document describes the embedded-side software driver which implements the serial interface between the application and the host PC. The serial interface covers the native UART/LPUART and SCI communication for all supported devices, the EOnCE/JTAG communication for the 56F8xxx family of hybrid MCUs, the CAN or LIN communication for the applicable devices, and the MQX_IO interface on the MQX RTOS OS platform.

This driver also supports the packet-driven BDM interface which enables the BDM interface to be used as the FreeMASTER communication device. The BDM enables non-intrusive access to the memory. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. To use more advanced FreeMASTER protocol features over the BDM interface, use the serial driver configured for the packet-driven mode.

1.1 Software

At the time of writing this document, the software driver supports the NXP products summarized in the table below. In the future, more platforms are to be supported by the driver. However, the approach to configure and use the driver must remain compatible with the one described in this document. See the release notes of the latest driver installation pack for an up-to-date list of supported devices, as well as the respective addendum to this user's guide.

Table 1-1. Supported platforms

CPU/platform	Tested with compiler	Sample applications available
56F8xxx Digital Signal Controllers (Hybrid Controllers)	CodeWarrior 10.6 Development Studio for MCU	Stand-alone applications created using the DSP56F800EX_Quick_Start r2.6 Boards: TWR-86F8200, TWR-86F8400
HC08/HCS08 MCUs	CodeWarrior 10.6 Development Studio for MCU	Applications created from the original CodeWarrior project stationery. Device: MC9S08JM60
HC12/HCS12 MCUs	CodeWarrior Development Studio for HC12 5.1	Applications created from the original CodeWarrior project stationery. Device: MC9S12DP256
HCS12X MCUs	CodeWarrior Development Studio for HCS12X 5.1	Applications created from the original CodeWarrior project stationery. Device: MC9S12XDP512 (both the banked and the large-data models)
MPC55xx Power Architecture MCUs	CodeWarrior Development Studio for MPC55xx V2.5	N/A
MPC56xx Power Architecture MCUs	CodeWarrior 10.6 Development Studio for MCU	Applications created from the original CodeWarrior project stationery for MPC5604P, MPC560xB/C, MPC564xL, MPC567xR, MPC560xE, MPC563xM, MPC564xA, and MPC560xS EVB.
MPC57xx Power Architecture MCUs	S32 Design Studio for Power Architecture v1.1	Applications created from the original S32 Design Studio project stationery for MPC577xK, MPC574xP, MPC577xM, MPC574xR, MPC577xC, MPC574xG, and MPC574xC EVB.

Table 1-1. Supported platforms (continued)

CPU/platform	Tested with compiler	Sample applications available
KEAxx MCUs	S32 Design Studio for ARM® Cortex®-M v1.2	Applications created from the original S32 Design Studio project stationery for KEAZ128 EVB.
S32Kxx MCUs	S32 Design Studio for ARM Cortex-M v1.2	Applications created from the original S32 Design Studio project stationery for S32K144 EVB.
S12Z MagniV MCUs	CodeWarrior 10.6 Development Studio for MCU	Applications created from the original CodeWarrior project stationery for S12ZVM EVB.
MCF51xx ColdFire Processors	CodeWarrior 10.6 Development Studio for MCU	Application created using the built-in CodeWarrior project-creation wizard. Board: EVB 51JM128
MCF52xx ColdFire Processors	CodeWarrior 10.6 Development Studio for MCU	Application created using the built-in CodeWarrior project-creation wizard. Board: TWR-MCF52259
Kinetis Cortex-M Processors	All as supported by KSDK1.1	Same as the KSDK Demo Applications Board: All as supported by KSDK1.1.
MQX Operating System (v4.1 or higher)	All as supported by MQX	N/A

1.2 Replacing existing drivers

The driver described in this document replaces the older drivers which were available separately for individual platforms, and were known as the PC Master SCI drivers. Because the FreeMASTER tool is fully compatible with the communication interface provided by the old PC Master drivers, it is strongly recommended to upgrade the existing embedded applications to the new FreeMASTER driver.

The main advantage of the new driver is the unification across all supported NXP processor products, as well as several new features that were added. One of the key features implemented in the new driver is the Target-Side Addressing (TSA) which enables an embedded application to describe the memory objects it grants the host access to. By enabling the “TSA-Safety” option, the application memory can be protected from illegal or invalid memory accesses.

1.3 Quick_Start tools

The NXP Quick_Start tools available for the MPC500/5500, DSP56F800, and 56F800E platforms also include some (older) versions of the FreeMASTER Serial Communication Driver. The stand-alone driver code described in this document is based on these individual Quick_Start drivers and may be used as their up-to-date replacement.

1.4 Kinetis SDK

The Kinetis SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with the demo applications for the NXP Kinetis family of MCUs. The FreeMASTER Serial Communication Driver package includes example applications for the latest public release of Kinetis SDK. The example applications are available in the form of a *.zip* file which must be extracted into the SDK installation.

NOTE

The FreeMASTER Serial Driver contains simple built-in device drivers for all communication peripheral modules and is not dependent on the Kinetis SDK drivers. However, the Kinetis SDK HAL drivers are used in the Kinetis example applications to initialize the clock subsystem and pin multiplexing and to set up the communication parameters (such as the baud rate, FIFO buffering, or signal polarity).

1.5 FreeMASTER Processor Expert component

Both the CodeWarrior and Kinetis Design Studio IDEs provided by NXP contain a graphical configuration extension called Processor Expert. The Processor Expert initialization components can be used to simplify the MCU device and peripheral configuration process and to make the application integration with the FreeMASTER serial driver easier. See the Peripheral Init Components and the FreeMASTER Component in the Processor Expert suite.

Chapter 2 DESCRIPTION

2.1 Introduction

This section shows how to add the FreeMASTER Serial Communication Driver into your application and configure and enable the connection to the FreeMASTER visualization tool.

2.2 Features

The FreeMASTER driver implements all the features necessary to establish the communication to the FreeMASTER visualization tool. The driver is a newly rewritten (backward compatible) version of the older PC Master driver, adding the support for new platforms, a better code structure and readability, and also new features such as the Pipes, Target-Side Addressing (TSA), TSA-Safety, and Application Command Callback functions.

The FreeMASTER protocol features:

- Read/write access to any memory location on the target.
- Atomic bit manipulation on the target memory (bit-wise write access).
- Oscilloscope access—optimized real-time access to up to eight variables. The sample rate depends on the communication speed.
- Recorder—the access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory (up to 64 KB).
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the variables, files, or hyperlinks exported by the target application.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol implementation.
- SCI as a native communication interface.
- EOnCE/JTAG as a communication interface on the 56F8xxx family of DSCs.
- CAN communication interface for platforms with the FlexCAN, MSCAN, or MCAN modules.
- LIN communication interface for the HCS12Z platform or other platforms where the standard LIN driver API is supported.
- MQX I/O communication interface. The standard MQX serial driver is used instead of the direct peripheral access.
- USB-CDC communication interface which uses the Kinetis SDK USB stack or the ColdFire/HCS08 Medical USB stack V3.0.
- Support for the ARM mbed™ development environment and the serial_t class.
- Ability to write-protect memory regions or individual variables.
- Ability to deny the access to the unsafe memory.
- C++ compatible.
- Two ways of handling the application commands:
  - Classic—the application polls the application command status to determine if any command is pending.

  - — Callback—the application registers a callback function which is automatically invoked upon the reception of a given command.
  - — The two approaches may be mixed in an application. The callback commands do not appear in the polling mechanism.
- The pipe callback handlers are invoked whenever new data are available for reading from the pipe.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control your embedded application.

2.2.1 Board Detection

The FreeMASTER protocol defines the standard way in which the host PC reads the platform-specific information needed to access the target resources. The board information includes these parameters:

- The version of the driver and the version of the protocol implemented.
- Driver name*.
- Target processor byte ordering (little/big endian).
- Communication buffer length.
- Address space granularity (1 B on most platforms; 2 B on 56F8xx DSP).
- Recorder capabilities*.

On smaller devices with limited flash and RAM memory resources, the board information may be restricted to a brief version by excluding the unnecessary items (the items marked by * in the list above).

2.2.2 Memory Read

This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The Variable Read feature is a slightly optimized variant of the Memory Read protocol feature. It enables reading 1-, 2-, or 4-byte variables while saving one byte on the communication line.

2.2.3 Memory Write

Similarly to the Memory Read operation, the Memory Write feature enables to write any RAM memory location on the target device. A single write command frame must fit into the target communication buffer, which is needed to split the large write requests into smaller ones.

A slightly optimized Variable Write variants exist to write the 2- and 4-byte variables.

2.2.4 Masked Memory Write

To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol. Except for the AND-mask applied to the data values being written, this feature is similar to a standard Memory Write.

A slightly optimized Masked Variable Write variants exist to write the 1- and 2-byte variables.

2.2.5 Oscilloscope

The protocol enables up to eight variables to be read at once on a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

This is an optional feature. If it is disabled, FreeMASTER uses the standard Memory Read or Variable Read accesses to read the graphed variables sequentially.

2.2.6 Recorder

The protocol defines a standard way of selecting up to eight variables on the target whose values are periodically recorded into the dedicated on-board memory buffer. After the data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

This is an optional feature and can be disabled if it is not needed in the application.

2.2.7 Fast Recorder

Similarly to the other features of the Serial Communication Driver, the Recorder feature is implemented in the C language and the code is common for all supported platforms. Such a general implementation brings some run-time overhead which can be unacceptable in applications where very short measuring cycles are required.

The Fast Recorder implementation is designed to address this requirement. The behavior of the Fast Recorder is similar to that of the standard Recorder, but it is implemented in the assembly language and optimized separately for different platforms. The code of the Fast Recorder sampling routine is split into a few atomic parts which can be selectively excluded to achieve a better performance. The following scenarios are possible with the Fast Recorder. The sampling times were measured with the 56F8013 test application running at 32 MHz with eight 16-bit variables sampled:

- The Fast Recorder is not used. The sampling time is approximately 27 μs.
- The Fast Recorder is used with the configurable variable count and addresses. The threshold-crossing trigger is not used. The sampling time is 3.5 μs.
- The Fast Recorder is used with the hardcoded variable addresses and no trigger capabilities. The sampling time 2.16 μs.

Same as with the standard Recorder feature, this is an optional feature and can be disabled. The Fast Recorder feature is supported only on the 56F8xxx platforms.

2.2.8 TSA

With the TSA feature, you can describe the variables and structure data types directly in the application source code and make this information available for the FreeMASTER tool. The tool can then use this information instead of reading it from the application’s ELF/Dwarf executable file.

The TSA feature enables you to create the so-called TSA tables, and put them directly into the embedded application. The TSA tables contain the descriptors of variables that you want to make visible to the host. The descriptors can describe the memory areas by specifying the address and size of the block (or more conveniently using the C variable names directly). You can also put the type information about the structures, unions, or arrays into the TSA table.

In the driver version 1.8.2, the new TSA table entries are supported to describe the memory-mapped files, virtual directories, or web URL hyperlinks. These new objects enable the target application to provide the whole graphical active content for the PC host-side tool.

2.2.9 TSA Safety

When the TSA is enabled in the application, the TSA Safety can be enabled and make the memory accesses validated directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

2.2.10 Application Commands

The Application Commands are high-level messages that the host may deliver to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After processing, when the embedded application acknowledges that the command is handled, the host receives the Result Code. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing.

2.2.11 Pipes

The Pipes enable the buffered stream-oriented data exchange between the PC Host and the target MCU. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losing any of them.

The Pipes require FreeMASTER version 2.0 (or higher) to be used on the PC Host computer.

2.3 Driver files

After installing the software, the driver source files can be found in the *driver* folder, further divided into the subfolders.

- *src_platforms* platform-specific folder—one folder exists for each supported processor platform (56F8xxx, HC08, HC12, MPC55xx, MPC56xx, MPC57xx, S32xx, KEAxx, Kxx, and so on). There is the *freemaster.h* master header file and the platform-dependent *.c* and header files. The *.c* file must be added to the project for compilation and linking. The *freemaster.h* file must be included in your application wherever you need to call any of the FreeMASTER driver API functions.
  - *freemaster.h*—master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
  - *freemaster_XXX.c* (XXX stands for platform identifier)—this file contains the platform-specific functions to access the data memory, communication buffer memory, serial interface interrupts, and other platform-specific code.
  - *freemaster_XXX.h* (XXX stands for the platform identifier)—this file defines the platform-specific SCI/CAN access macros, memory access inline functions, and other platform-specific declarations.
  - *freemaster_fastrec.c*—Fast Recorder implementation for a given platform.

  - *freemaster_fastrec.h*—Fast Recorder header file which is indirectly included into the end application when the Fast Recorder is enabled. This file contains the inline assembly code and macros used to compose the fast sampling code.
  - *freemaster_cfg.h.example*—this file can serve as an example of the FreeMASTER driver configuration file. Save this file into your project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get your project-specific configuration options, and to optimize the compilation of the driver.
- *src_common* folder—contains the common driver source files shared by the driver for all supported platforms. All *.c* files must be added to your project, compiled, and linked together with your application.
  - *freemaster_serial.c*—implements the serial communication buffers, FIFO queuing, and other communication-related operations. This file uses the SCI or JTAG module access macros from the platform-dependent header file (see above).
  - *freemaster_can.c*—implements the FlexCAN/MSCAN/MCAN communication and other communication-related operations. This file uses the FlexCAN, MSCAN, or MCAN module access macros from the platform-dependent header file (see above).
  - *freemaster_lin.c*—implements the LIN communication using the standard LIN Transport Layer API, as defined for the LIN Diagnostic Class III.
  - *freemaster_bdm.c*—implements the packet-driven BDM communication buffer and other communication-related operations.
  - *freemaster_protocol.c*—implements the FreeMASTER protocol decoder and handles the basic Memory Read or Memory Write commands.
  - *freemaster_rec.c*—handles the Recorder-specific commands and implements the Recorder sampling routine. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
  - *freemaster_scope.c*—handles the Oscilloscope-specific commands. In case the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles void.
  - *freemaster_pipes.c*—implements the Pipes functionality when the Pipes feature is enabled.
  - *freemaster_appcmd.c*—handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. See *FreeMASTER for Embedded Applications* (document FMSTERUG) for more details on this feature. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
  - *freemaster_tsa.c*—handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. In case the TSA is disabled by the FreeMASTER driver configuration file, this file compiles void.
  - *freemaster_private.h*—contains the declarations of functions and data types used internally in the driver. Based on the platform-identification macro declared in *freemaster.h*, this file includes a platform-dependent header file (*freemaster_XXX.h*). The *freemaster_private.h* file also contains the C pre-processor statements to perform intensive compile-time verification of the driver configuration.
  - *freemaster_defcfg.h*—defines all FreeMASTER options to the default value when the option is not set in the *freemaster_cfg.h* file.
  - *freemaster_protocol.h*—defines the FreeMASTER protocol constants used internally by the driver.

— *freemaster_tsa.h*—contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (from *freemaster.h*).
— *freemaster_rec.h*—exports the global variables of the Recorder module. It is used internally by the Fast Recorder implementation, if this is enabled.

NOTE

The header files from the *src_common* folder are private to the driver and must not be included anywhere in your application. Similarly to the platform-dependent folder, add the *src_common* folder to your compiler's "include" search path.

2.4 Driver configuration

The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with your other project source files. The FreeMASTER driver source files include the *freemaster_cfg.h* file, and use macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, this can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file from the platform-specific source folder, rename it to *freemaster_cfg.h*, and save it into the project area.

NOTE

It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at the project-specific location, so that it does not affect the other applications that use the driver.

2.4.1 Processor Expert configuration tool

Using FreeMASTER in applications created using Processor Expert is very easy thanks to a special FreeMASTER component. In the component catalogue, locate the FreeMASTER component and add it to the project. The component enables you to set up both the communication parameters and FreeMASTER-specific options. All the configurable items described in the next section are graphically configurable from within the Processor Expert component wizard.

2.4.2 Quick_Start graphical configuration tool

When using the FreeMASTER driver in an application based on the Quick_Start project, the *freemaster_cfg.h.quickstart* can be renamed to *freemaster_cfg.h* and used as a configuration file placeholder. The *freemaster_cfg.h.quickstart* file only includes the *appconfig.h* file and other Quick_Start-specific header files which enable the FreeMASTER driver to be configured using the Quick_Start graphical configuration tool.

Working in the Quick_Start environment and using its graphical configuration tool helps to easily configure all processor peripheral modules in a user-friendly graphical application. This also includes the configuration of the system clocks, the SCI module, and the MSCAN/FlexCAN/MCAN module required by the FreeMASTER driver, as well as the FreeMASTER driver itself. All configurable items described in Section 2.4.3, "Configurable items" are graphically configurable.

2.4.3 Configurable items

The following table describes the *freemaster_cfg.h* configuration options which are common to all supported platforms. See Chapter 4, "PLATFORM-SPECIFIC TOPICS" for a detailed description of the platform-specific options.

Table 2-1. Driver configuration options

Statement	Values	Description
#define FMSTR_LONG_INTR #define FMSTR_SHORT_INTR #define FMSTR_POLL_DRIVEN	boolean (0 or 1) boolean (0 or 1) boolean (0 or 1)	Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See Section 2.4.4, "Driver interrupt modes". FMSTR_LONG_INTR—long interrupt mode FMSTR_SHORT_INTR—short interrupt mode FMSTR_POLL_DRIVEN—poll-driven mode **Note:** This option is not supported by the MQX_IO interface. The FMSTR_POLL_DRIVEN option is not supported by the USB_CDC interface.
#define FMSTR_DISABLE	boolean (0 or 1)	Define it as non-zero when you want to disable all FreeMASTER features. The default value is “false”.
#define FMSTR_USE_SCI	boolean (0 or 1)	Define it as non-zero when you want to communicate over the SCI or the standard UART interface. This interface is also good for the Kinetis LPSCI module. The default value is “true” when the SCI base address is defined.
#define FMSTR_USE_LPUART	boolean (0 or 1)	Define it as non-zero when you want to communicate over the Kinetis LPUART interface. The default value is “false”.
#define FMSTR_USE_ESCI	boolean (0 or 1)	Define it as non-zero when you want to communicate over the Power Architecture ESCI interface. The default value is “false”.
#define FMSTR_SCI_BASE	address	Specify the base address of the SCI, LPUART, or ESCI peripheral module to be used for the communication. If you use the symbolic name, make sure to also include the header file where the symbol is defined or declared.
#define FMSTR_SCI_BASE_DYNAMIC	boolean (0 or 1)	Define it as non-zero if the SCI, LPUART, or ESCI module base address is to be specified dynamically in the application runtime by the `FMSTR_SetSciBaseAddress()` function. The default value is “true” when the FMSTR_SCI_BASE is not defined. Otherwise it is “false”.
#define FMSTR_COMM_BUFFER_SIZE	0...255	Specify the size of the communication buffer to be allocated by the driver. When undefined or defined to zero (recommended), the buffer size will be automatically calculated in *freemaster_private.h*, based on the driver features selected. The default value is “0” (automatic).
#define FMSTR_COMM_RQUEUE_SIZE	0...255	Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode. The default value is 32 B.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#define FMSTR_USE_MSCAN	boolean (0 or 1)	Define it as non-zero when you want to communicate over the MSCAN (Multi-Scalable Controller Area Network) instead of the SCI or JTAG. Note that a special CAN communication plugin for FreeMASTER is required on the PC side. The default value is “false”.
#define FMSTR_USE_FLEXCAN	boolean (0 or 1)	Define it as non-zero when you want to communicate over the FlexCAN instead of the SCI or JTAG. Note that a special CAN communication plugin for FreeMASTER is required on the PC side. The default value is “false”.
#define FMSTR_USE_FLEXCAN32	boolean (0 or 1)	Define it as non-zero when you want to communicate over the FlexCAN on the 56F84xxx devices. Note that a special CAN communication plugin for FreeMASTER is required on the PC side. The default value is “false”. **Note:** This configuration option is supported by the 56F8xxx platform.
#define FMSTR_USE_MCAN	boolean (0 or 1)	Define it as non-zero when you want to communicate over the MCAN (Modular Controller Area Network) instead of the SCI or JTAG. Note that a special CAN communication plugin for FreeMASTER is required on the PC side. The default value is “false”. **Note:** This configuration option is supported by some of the the MPC57xx platforms.
#define FMSTR_CAN_BASE	address	Specify the base address of the MSCAN/FlexCAN/MCAN peripheral module to be used for communication. If you use the symbolic name, make sure to also include the header file where the symbol is defined or declared.
#define FMSTR_CAN_BASE_DYNAMIC	boolean (0 or 1)	Define it as non-zero if the CAN module base address is to be specified dynamically in the application runtime by the `FMSTR_SetCanBaseAddress()` function. The default value is “true” when FMSTR_CAN_BASE is not defined. Otherwise it is “false”.
#define FMSTR_FLEXCAN_TXMB	0...15 (0...31)	Define the FlexCAN message buffer for the TX communication. The default value is “0”.
#define FMSTR_FLEXCAN_RXMB	0...15 (0...31)	Define the FlexCAN message buffer for the RX communication. The default value is “1”.
#define FMSTR_USE_LINTL	boolean (0 or 1)	Define it as non-zero when you want to communicate over the LIN Transport Layer protocol API. A LIN driver with the standard Transport Layer API must be compiled in the application too. Note that a special LIN communication plug-in for the FreeMASTER tool is required on the PC side. The default value is “false”.
#define FMSTR_USE_USB_CDC	boolean (0 or 1)	Define it as non-zero when you want to communicate over the virtual serial line implemented as a USB-CDC device. A compatible USB driver must be compiled in the application too. The default value is “false”.
#define FMSTR_USB_LEGACY_STACK	boolean (0 or 1)	Define it as non-zero when using the USB-CDC class implementation of the legacy NXP Medical USB stack v3.0. Leave it undefined when using the USB stack included in Kinetis SDK v1.1 (or higher). The default value is “false”.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#define FMSTR_USE_PDBDM	boolean (0 or 1)	Define it as non-zero when you want to communicate over the packet-driven BDM instead of the SCI, JTAG, or CAN interface. Note that this interface supports all FreeMASTER features in the same way as the SCI or CAN interfaces do. The default value is “false”.
#define FMSTR_USE_READMEM	boolean (0 or 1)	Define it as non-zero to implement the Memory Read command (recommended). The default value is “true”.
#define FMSTR_USE_READVAR	boolean (0 or 1)	Define it as non-zero to implement the Variable Read command. The default value is “false”.
#define FMSTR_USE_WRITEMEM	boolean (0 or 1)	Define it as non-zero to implement the Memory Write command (recommended). The default value is “true”.
#define FMSTR_USE_WRITEVAR	boolean (0 or 1)	Define it as non-zero to implement the Variable Write command. The default value is “false”.
#define FMSTR_USE_WRITEMEMMASK	boolean (0 or 1)	Define it as non-zero to implement the Masked Memory Write command (recommended). The default value is “true”.
#define FMSTR_USE_WRITEVARMASK	boolean (0 or 1)	Define it as non-zero to implement the Masked Variable Write command. The default value is “false”.
#define FMSTR_BYTE_BUFFER_ACCESS	boolean (0 or 1)	Define it as non-zero to enable the byte access to the communication buffer. The default value is “true”. **Note:** This configuration option is supported by the Kinetis platform. All Cortex M0-based devices require this option to be set to avoid misaligned access to the integer parameters, which is not supported on this platform.
#define FMSTR_ISR_CALLBACK	string	Define the name of the callback function. This feature enables calling the application callback function in the interrupt communication handler. **Note:** This configuration option is supported by the MQX platform.
#define FMSTR_DEBUG_TX	boolean (0 or 1)	Define it as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings. The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+©W) which are easy to capture using the serial terminal tools. This feature requires the `FMSTR_Poll()` function to be called periodically. The default value is “false”.
Oscilloscope		
#define FMSTR_USE_SCOPE	boolean (0 or 1)	Define it as non-zero to implement the Oscilloscope feature. The default value is “false”.
#define FMSTR_MAX_SCOPE_VARS	2...8	The number of variables to be supported in the Oscilloscope feature.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
Recorder		
#define FMSTR_USE_RECORDER	boolean (0 or 1)	Define it as non-zero to implement the Recorder feature. The default value is “false”.
#define FMSTR_MAX_REC_VARS	2...8	The number of variables to be supported in the Recorder feature.
#define FMSTR_REC_OWNBUFF	boolean (0 or 1)	When true (non-zero), the driver does not allocate the recorder buffer. Call the `FMSTR_SetUpRecBuff` function to configure the recorder buffer before using it. The default value is “false” (the driver allocates the buffer).
#define FMSTR_REC_BUFF_SIZE	16bit number	When FMSTR_REC_OWNBUFF is “false”, this constant defines the size of the recorder buffer to be allocated by the driver.
#define FMSTR_REC_TIMEBASE	16bit number	This constant tells the host how often you call the recorder sampling routine (`FMSTR_Recorder`) in your application. The host uses this information to draw the x-axis of the recorder graph properly. Use 0 as “unknown” (the x-axis is drawn indexed-only). Use one of the following macros to specify the time value (x is a 14-bit constant): FMSTR_REC_BASE_SECONDS(x) FMSTR_REC_BASE_MILLISEC(x) FMSTR_REC_BASE_MICROSEC(x) FMSTR_REC_BASE_NANOSEC(x)
#define FMSTR_REC_FLOAT_TRIG	boolean (0 or 1)	Define it as non-zero to implement the floating-point triggering. FreeMASTER v1.3.12 (or higher) is required. The default value is “false”.
Fast Recorder		
#define FMSTR_USE_FASTREC	boolean (0 or 1)	Define it as non-zero to implement the Fast Recorder feature. This also requires the Recorder feature to be enabled by setting FMSTR_USE_RECORDER to non-zero. The default value is “false”. **Note:** This configuration option is supported by the 56F8xxx platform.
Application Commands		
#define FMSTR_USE_APPCMD	boolean (0 or 1)	Define it as non-zero to implement the Application Commands feature. The default value is “false”.
#define FMSTR_APPCMD_BUFF_SIZE	1...255	The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.
#define FMSTR_MAX_APPCMD_CALLS	0...255	The number of different Application Commands that you can assign to the callback handler function (see `FMSTR_RegisterAppCmdCall`). The default value is “0”.
Target-Side Addressing		
#define FMSTR_USE_TSA	boolean (0 or 1)	Define it as non-zero to implement the TSA feature. The default value is “false”.

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#define FMSTR_USE_TSA_SAFETY	boolean (0 or 1)	Enable the memory access validation in the FreeMASTER driver. The host can't access the memory not described by any TSA descriptor. The write access is denied for the read-only objects.
#define FMSTR_USE_TSA_INROM	boolean (0 or 1)	Declare all TSA descriptors as "const", which enables the linker to put the data into the flash memory. The actual result depends on your linker settings or the linker commands used in your project. The default value is "false".
#define FMSTR_USE_TSA_DYNAMIC	boolean (0 or 1)	Enable the runtime-defined TSA entries to be added to TSA table by the `FMSTR_TsaAddVar()` function. The default value is "false".
Pipes		
#define FMSTR_USE_PIPES	boolean (0 or 1)	Enable the FreeMASTER Pipes feature to be used. The default value is "false".
#define FMSTR_MAX_PIPES_COUNT	1...63	The number of simultaneous pipe connections to support. The default value is "1".
Advanced Options		
#define FMSTR_USE_NOEX_CMDS	boolean (0 or 1)	Enable the 16-bit addressing commands in the FreeMASTER driver. The FreeMASTER protocol commands carrying 16-bit addresses are also called standard or non-EX. See the note in the parameter below.
#define FMSTR_USE_EX_CMDS	boolean (0 or 1)	Enable the EX FreeMASTER protocol commands to be processed by the driver. The EX commands differ from the standard commands only in using 32-bit addresses. The EX commands are always two bytes longer than their standard counterparts. **Note:** Although being configurable items, it is not recommended to specify these two parameters in the configuration file. The default values of both options are defined in the platform-dependent header file and it should not be necessary to override them manually. See Chapter 4, "PLATFORM-SPECIFIC TOPICS" for more details about the individual platforms.
Light version (achieving smaller RAM/flash footprint)		
#define FMSTR_LIGHT_VERSION	boolean (0 or 1)	Define it as non-zero to activate the light version of the FreeMASTER driver. The light version of FreeMASTER uses less RAM and flash memory and enables you to limit the FreeMASTER functionality. The default value is "false".
#define FMSTR_SCI_TWOWIRE_ONLY	boolean (0 or 1)	Define it as non-zero to limit the SCI communication to the two-wire configuration. The default value is "FMSTR_LIGHT_VERSION".
#define FMSTR_REC_COMMON_ERR_ CODES	boolean (0 or 1)	Define it as non-zero to change the Recorder initialization and communication error codes to one common code. The default value is "FMSTR_LIGHT_VERSION".

Table 2-1. Driver configuration options (continued)

Statement	Values	Description
#define FMSTR_REC_STATIC_ POSTTRIG	0...32000	Specify the hardcoded constant number of the post-trigger samples in the Recorder function. This value represents the value of samples to be acquired after the trigger event and can't be changed by FreeMASTER. For a standard functionality where the pre-trigger samples are set by the FreeMASTER tool, define this value as "0". The default value is "0". **Note:** Keep this value lower than the "Recorded samples" value configured in the FreeMASTER tool.
#define FMSTR_REC_STATIC_ DIVISOR	0...32767	Specify the hardcoded constant number of the Recorder time divisor in the Recorder function. This value represents the time base multiple of the sampling period. For a standard functionality where the time base multiples are set by the FreeMASTER tool, define this value as "0". To completely disable the time-divisor functionality, set this value to "1". The default value is "0".

2.4.4 Driver interrupt modes

To implement the serial communication, the FreeMASTER driver handles the SCI/UART or CAN module receive and transmit requests. Select whether the driver processes the SCI/UART or CAN communication automatically as an interrupt service routine or if it only polls the status of the module (typically during the application idle time). See this table for a description of each mode:

Table 2-2. Driver interrupt modes

Mode	Description
Completely Interrupt-Driven (FMSTR_LONG_INTR = 1)	Both the serial (SCI/CAN) communication and the FreeMASTER protocol decoding and execution is done in the `FMSTR_Isr` interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority. The application must subscribe the `FMSTR_Isr` function as the serial interface interrupt vector (or multiple vectors when the serial interface receive, transmit, and/or communication errors are handled by different interrupts). **Note1:** The MQX_IO interface can't open the communication interface in the interrupt mode by this option. **Note2:** The USB CDC interface executes all communications in the `USB_Isr` interrupt service routine. **Note3:** The BDM and Packet-Driven BDM interfaces don't support the interrupt mode.
Mixed Interrupt and Polling Modes (FMSTR_SHORT_INTR = 1)	The raw serial (SCI/CAN) communication is handled by the `FMSTR_Isr` interrupt service routine, while the protocol decoding and execution is handled by the `FMSTR_Poll` routine. Call the `FMSTR_Poll` during the idle time in the application main loop. The interrupt processing is relatively fast and deterministic. Upon an SCI receive event, the received character is only placed into a FIFO-like queue and is not further processed. Upon a CAN receive event, the received packet is stored into the receive buffer. When transmitting, the characters are just fetched from the prepared transmit buffer. The application must subscribe the `FMSTR_Isr` function as the serial interface interrupt vector (or multiple vectors in case the serial interface receive, transmit, and/or communication errors are handled by different interrupts). When the SCI is used as a serial communication interface, ensure that the `FMSTR_Poll` function is called at least once per N character time periods. N is the length of the FreeMASTER FIFO queue (FMSTR_COMM_RQUEUE_SIZE) and the character time is the time needed to transmit or receive a single byte over the SCI line. **Note1:** The MQX_IO interface can't open the communication interface in the interrupt mode by this option. **Note2:** The USB CDC interface executes all communications in the `USB_Isr` interrupt service routine. **Note3:** The BDM and Packet-Driven BDM interfaces do not support the interrupt mode.
Completely Poll-driven (FMSTR_POLL_DRIVEN = 1)	Both the serial (SCI/CAN) communication and the FreeMASTER protocol execution are done in the `FMSTR_Poll` routine. No interrupts are needed and the `FMSTR_Isr` code compiles to an empty function. When using this mode, ensure that the `FMSTR_Poll` function is called by the application at least once per the SCI character time which is the time needed to transmit or receive a single character. **Note:** This poll mode is not supported by the FMSTR_USE_USB_CDC interface.

In the latter two modes (FMSTR_SHORT_INTR, FMSTR_POLL_DRIVEN), the protocol handling takes place in the `FMSTR_Poll` routine. An application interrupt can occur in the middle of the Read Memory or Write Memory commands execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, it is not recommended to use FreeMASTER to visualize or monitor the volatile variables (those being modified anywhere in the user interrupt code).

The same restriction applies even in the full interrupt mode (FMSTR_LONG_INTR) if the volatile variables are modified in the interrupt code of priority higher than the priority of the SCI/CAN interrupt.

2.5 Data types

Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the *freemaster.h* master header file and the other private data types are defined in the platform-specific header file.

To prevent name conflicts with the symbols used in your application, all data types, macros, and functions have the `FMSTR_` prefix. There are no global variables used in the driver. The private variables are all declared as static and are preceded either with the `fmstr_` prefix or the `pcm_` prefix.

2.6 Embedded communication interface initialization

The FreeMASTER driver does not perform any initialization or configuration of the SCI/CAN module it uses to communicate. It is your responsibility to configure the communication module before the FreeMASTER driver is initialized by the `FMSTR_Init` call.

When the SCI module is used as the FreeMASTER communication interface, configure the SCI receive and transmit pins, the serial communication baud rate, the parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or short interrupt modes of the driver (see Section 2.4.4, "Driver interrupt modes"), configure the interrupt controller and set the `FMSTR_Isr` function as the SCI interrupt service routine.

When the CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or short interrupt modes of the driver (see Section 2.4.4, "Driver interrupt modes"), configure the interrupt controller and set the `FMSTR_Isr` function as the CAN interrupt service routine.

When the USB CDC interface is used as the FreeMASTER communication interface, configure the USB module clock to 48 MHz, set the `USB_Isr` as the USB interrupt service routine, and enable the USB interrupt vector after the `FMSTR_Init()` call.

On some NXP platforms, there are user-friendly graphical tools that enable a convenient way to configure the processor peripheral modules.

NOTE

It is not necessary to enable or unmask the SCI interrupts (TIE and RIE bits), nor it is necessary to activate the SCI receiver or transmitter engines (TE and RE bits) before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines as required during the runtime.

2.7 FreeMASTER Recorder calls

When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER), call the `FMSTR_Recorder` function periodically in places where the data recording occurs. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere you want to sample the variable values. The example applications provided together with a driver code call the `FMSTR_Recorder` in the main application loop.

In applications where you call the `FMSTR_Recorder` equidistantly, use the FMSTR_REC_TIMEBASE macro to communicate the Recorder sampling period to the host application. If used in this way, the

FreeMASTER Recorder displays the X-axis of the Recorder graph properly recalculated into the time domain.

See Section 3.3, "Fast Recorder API" for information on how to call the Fast Recorder sampling routine.

2.8 Driver usage

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all *.c* files of the FreeMASTER driver from the *src_common* and *src_platforms/#your_platform#* folders are a part of your compiler project. See Section 2.3, "Driver files" for details.
- Configure the FreeMASTER driver by creating or editing the *freemaster_cfg.h* file and saving it into your project directory. See Section 2.4, "Driver configuration" for details.
- Include the *freemaster.h* file into any application source file that makes the FreeMASTER API calls.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, route the SCI (JTAG or CAN) interrupts to the `FMSTR_Isr` function and set the interrupt priority levels (if applicable).
- Initialize the SCI, JTAG, or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the SCI or CAN interrupts.
- Call the `FMSTR_Init` function.
- In the main application loop, make sure to call the `FMSTR_Poll` API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally for the CPU. You don’t have to enable the SCI or CAN interrupts individually; this is handled by the driver itself.

2.9 Communication troubleshooting

The most common problem that causes the communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When the communication between the PC Host running FreeMASTER and the target MCU can’t be established, try enabling the FMSTR_DEBUG_TX option in the *freemaster_cfg.h* file and make sure to call the `FMSTR_Poll()` function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame on the SCI/UART or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Chapter 3 DRIVER API

This section describes the driver Application Programmer's Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

3.1 Control API

There are three key functions to initialize and use the driver.

3.1.1 FMSTR_Init

Prototype

```
void FMSTR_Init(void)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_protocol.c
```

Description

This function initializes the internal variables of the FreeMASTER driver and enables the communication interface (SCI, JTAG, or CAN). This function does not change the configuration of the selected communication module. The module must be initialized before the `FMSTR_Init` function is called.

The call to this function must occur before calling any other FreeMASTER driver API functions.

3.1.2 FMSTR_Poll

Prototype

```
void FMSTR_Poll(void)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_serial.c, freemaster_can.c or freemaster_bdm.c
(depending on communication interface used)
```

Description

In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see Section 2.4.4, "Driver interrupt modes"). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the `FMSTR_Poll` function is called during the "idle" time in the main application loop.

To prevent the receive data overflow (loss) for the SCI communication interface, make sure that the `FMSTR_Poll` function is called at least once per the time calculated as:

$$N * T_{char}$$

where:

- *N* is equal to the length of the receive FIFO queue (configured by the FMSTR_COMM_RQUEUE_SIZE macro). *N* is 1 for the poll-driven mode.

- T_{char} is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

NOTE

In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different modes only by changing the configuration macros in the *freemaster_cfg.h* file.

3.1.3 FMSTR_Isr

Prototype

```
void FMSTR_Isr(void)
```

Declaration

```
freemaster.h
```

Implementation

```
platform-dependent C file (src_platforms directory)
```

Description

This is the interface interrupt service routine of the FreeMASTER driver. In the long or short interrupt modes (see Section 2.4.4, "Driver interrupt modes"), this function must be set as the interface interrupt vector. On platforms where the interface processing is split into multiple interrupts, this function must be set as a vector for each such interrupt.

NOTE

In a completely poll-driven mode, this function is compiled as an empty function and does not have to be used.

3.2 Recorder API

3.2.1 FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(void)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_rec.c
```

Description

This function takes one sample of the variables being recorded using the FreeMASTER Recorder. If the Recorder is not active when the `FMSTR_Recorder` function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger condition is evaluated.

If the trigger condition is satisfied, the recorder enters the post-trigger mode, where it counts the follow-up samples (`FMSTR_Recorder` function calls) and de-activates the Recorder when the required post-trigger samples are sampled.

The `FMSTR_Recorder` function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

3.2.2 FMSTR_TriggerRec

Prototype

```
void FMSTR_TriggerRec(void)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_rec.c
```

Description

This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application when you want to have the trigger occurrence under your control.

3.2.3 FMSTR_SetUpRecBuff

Prototype

```
void FMSTR_SetUpRecBuff(FMSTR_ADDR nBuffAddr, FMSTR_SIZE nBuffSize)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_rec.c
```

Arguments

nBuffAddr (in)—pointer to the memory to be used as the Recorder buffer

nBuffSize (in)—size of the memory buffer (64 KB maximum)

Description

This function can only be used when the FMSTR_REC_OWNBUFF configuration constant is set to a non-zero value. Call this function to “give” the data buffer that you allocated to the FreeMASTER driver which uses it as the Recorder buffer.

A buffer of up to 64 KB may be used as the Recorder buffer.

3.3 Fast Recorder API

When the Fast Recorder feature is enabled in the `freemaster_cfg.h` file, you can invoke the fast sampling routine from within the application. By default, the use of the Fast Recorder is fully transparent and a call to the `FMSTR_Recorder` function is automatically substituted with the `FMSTR_FastRecorder` inline function call.

The main differences between the standard Recorder and the Fast Recorder are:

- The number of the Recorder variables is configurable from the PC-side FreeMASTER tool in both Recorder implementations.
- The Fast Recorder may optionally use a hardcoded list of variables to achieve a better performance.
- The addresses of variables being recorded are configurable from the PC-side in both Recorder implementations. The Fast Recorder may optionally use a hardcoded list of variables to achieve a better performance.
- The sizes of the variables being recorded are typically not configurable in the Fast Recorder implementation.
- A general threshold-crossing trigger condition is not available in the Fast Recorder implementation. Hardcode your own trigger condition into the application code and pass the condition boolean result to the `FMSTR_FastRecorder` call. The other way to trigger the Recorder from within the application is to call the `FMSTR_TriggerRec` function.

Except for the `FMSTR_FastRecorder` function, the Fast Recorder API is only needed when more drastic optimizations are to be hardcoded into the application. For example, the Recorder can be configured for the hardcoded variable addresses and the trigger mechanism may be simplified or even completely excluded.

3.3.1 FMSTR_FastRecorder

Prototype

```
inline void FMSTR_FastRecorder(FMSTR_BOOL bTriggerNow)
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Implementation

```
freemaster_fastrec.h (inline part)
freemaster_fastrec.c (static code part)
```

Arguments

bTriggerNow (in)—when TRUE, the recorder trigger condition is set as satisfied

Description

Similarly to the `FMSTR_Recorder` function, this function takes one sample of the Recorder variables and saves it into the internal circular buffer. This function is normally called automatically instead of the `FMSTR_Recorder` function when the Fast Recorder feature is enabled, but it may be also called explicitly.

The explicit calling of this function is the only way to enable the trigger condition evaluation with the Fast Recorder. When the triggering is required, the application evaluates the trigger condition itself and passes a boolean result to the `FMSTR_FastRecorder` function call. When the parameter value is true, the sampling is automatically set to the post-trigger count-down state, the remaining samples are taken, and the PC is then notified that the Recorder is done.

The parameter value is ignored when the Recorder is already in the post-trigger mode.

By default, the normal trigger handling is implemented by the Fast Recorder implementation. This means that the trigger must be first armed by having the condition not-satisfied (parameter is set to false) and only then the sampling may be triggered by setting the parameter to true. Using the Fast Recorder API, the trigger evaluation may be either completely excluded or set to the faster simple handling mode in which the arming is not required.

3.3.2 Fast Recorder inline code macros

Instead of calling the `FMSTR_FastRecorder` function, you may achieve a yet better optimization using the Fast Recorder API macros. The macros must appear in the code in a given order (described below). The macros must not be interlaced with any user code.

These Fast Recorder macros are defined and they must appear in the code in this order:

- Recorder code block begin—select when the sampling frequency divider is applied:
  - `FMSTR_FASTREC_BEGIN(do_fdiv)`
- Variable selection—either configurable or hardcoded addresses:
  - `FMSTR_FASTREC_STDVARS()`
  - `FMSTR_FASTREC_VARxx(variable_name)`
- Trigger code inlining:
  - `FMSTR_FASTREC_TRIGGER_FULL(cond_result)`
  - `FMSTR_FASTREC_TRIGGER_SIMPLE(cond_result)`
  - `FMSTR_FASTREC_TRIGGER_IMMEDIATE(cond_result)`
  - `FMSTR_FASTREC_TRIGGER_VOID()`
- Recorder code end:
  - `FMSTR_FASTREC_END()`

3.3.2.1 FMSTR_FASTREC_BEGIN

Macro declaration

`FMSTR_FASTREC_BEGIN(`*`do_fdiv`*`)`

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

do_fdiv (in)—when TRUE, the recorder applies the configurable frequency divider

Description

This macro begins to inline the Fast Recorder sampling code. When the `do_fdiv` parameter is false, the code for the frequency divider sampling is skipped.

3.3.2.2 FMSTR_FASTREC_STDVARS

Macro declaration

`FMSTR_FASTREC_STDVARS()`

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

None

Description

This macro inlines the code which takes one sample for each variable configured from the PC-side FreeMASTER tool. Using this macro, the addresses and count of the recorded variables are configurable in the same way as with the standard Recorder.

Typically, there is a limitation in the variable sizes that can be recorded using the Fast Recorder. For example, the 56F8xxx platform enables only the 16-bit variables to be sampled.

3.3.2.3 FMSTR_FASTREC_VARxx

Macro declaration

```
FMSTR_FASTREC_VARxx()
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

None, the xx suffix identifies the bit-size of the hardcoded variable to be sampled

Description

This macro helps to hardcode the variable addresses that are sampled. Using this macro, the PC-side FreeMASTER tool can't change the variable count and addresses. Do not use this macro together with the `FMSTR_FASTREC_STDVARS` macro.

To achieve correct results, configure the PC-side Recorder for the same number of variables of the same size as those hardcoded in the application.

3.3.2.4 FMSTR_FASTREC_TRIGGER_FULL

Macro declaration

```
FMSTR_FASTREC_TRIGGER_FULL(cond_result)
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

***cond_result*—**trigger condition result, as evaluated by the application

Description

This macro inlines the standard trigger-handling chunk into the recording code. With the Fast Recorder, the inlined code does not evaluate the trigger condition itself, but it accepts a boolean parameter which carries the trigger information from the calling application.

With the _FULL macro, the trigger must be armed first by passing the `cond_result` value set to `false` before the trigger may occur. When the standard threshold-crossing trigger conditions are used, the arming makes sure that the Recorder is not triggered by a simple value comparing. It detects and waits for a real threshold crossing.

3.3.2.5 FMSTR_FASTREC_TRIGGER_SIMPLE

Macro declaration

```
FMSTR_FASTREC_TRIGGER_SIMPLE(cond_result)
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

***cond_result*—**trigger condition result, as evaluated by the application

Description

This macro inlines the standard trigger-handling chunk into the recording code. With the Fast Recorder, the inlined code does not evaluate the trigger condition itself, but it accepts a boolean parameter which carries the trigger information from the calling application.

With the _SIMPLE macro, the trigger does not need to be armed before triggering. This makes the code faster, but it does not detect the threshold-crossing condition properly (if such condition is used in the application).

3.3.2.6 FMSTR_FASTREC_TRIGGER_IMMEDIATE

Macro declaration

```
FMSTR_FASTREC_TRIGGER_IMMEDIATE(cond_result)
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

None

Description

This macro inlines the fastest trigger-handling chunk possible into the recording code. With the Fast Recorder, the inlined code does not evaluate the trigger condition itself, but it accepts a boolean parameter which carries the trigger information from the calling application.

With the _IMMEDIATE macro, when the `cond_result` parameter is true, the Recorder is stopped immediately, without even finishing the post-trigger recording. This results in the Recorder with a 100 % pre-trigger positioning within the sampling buffer.

3.3.2.7 FMSTR_FASTREC_TRIGGER_VOID

Macro declaration

```
FMSTR_FASTREC_TRIGGER_VOID()
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

None

Description

This macro is just a placeholder for the trigger inline code and compiles to an empty code.

3.3.2.8 FMSTR_FASTREC_END

Macro declaration

```
FMSTR_FASTREC_END()
```

Declaration

```
freemaster_fastrec.h use freemaster.h
```

Arguments

None

Description

This macro finishes the inlining of the Fast Recorder sampling code.

3.4 TSA API

When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), define the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always at least one TSA Table List defined which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. After this, you can create the FreeMASTER variables based on these symbols. The TSA is supported in FreeMASTER version 1.2.39 (or higher).

3.4.1 TSA table definition

The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide you with the access to tthe individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro:

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

Where `table_id` is any valid C-language symbol identifying the table. There can be any number of TSA tables in the application.

After this opening macro, the TSA descriptors are placed using these macros:

```
FMSTR_TSA_RW_VAR(name, type)                        // read/write variable entry
FMSTR_TSA_RO_VAR(name, type)                        // read-only variable entry
FMSTR_TSA_STRUCT(struct_name)                       // structure type entry
FMSTR_TSA_MEMBER(struct_name, member_name, type)    // structure member entry
FMSTR_TSA_RW_MEM(name, type, address, size)         // read/write memory block
FMSTR_TSA_RO_MEM(name, type, address, size)         // read-only memory block
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

The TSA descriptor macros accept these parameters:

- `name`—the variable name. The variable must be defined before the TSA descriptor references it.
- `type`—the variable or member type. Only one of the pre-defined type constants may be used, as described in Table 3-1.
- `struct_name`—the structure type name. The type must be defined (typedef) before the TSA descriptor references it.
- `member_name`—the structure member name, without the dot at the beginning. The parent structure name is specified as a separate parameter in the FMSTR_TSA_MEMBER descriptor.

NOTE

The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table. To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file. Despite its name, the FMSTR_TSA_STRUCT macro may be also used to describe the union data types.

Table 3-1. TSA type constants

Constant	Description
FMSTR_TSA_UINT8 FMSTR_TSA_UINT16 FMSTR_TSA_UINT32 FMSTR_TSA_UINT64	1-, 2-, 4-, or 8-byte unsigned integer type. Use it for both the standard C-language types (unsigned char, unsigned short, or unsigned long) and the user-defined integer types commonly used on different platforms (UWord8, UWord16, uint8_t, unit16_t, Byte, Word, LWord, and other).
FMSTR_TSA_SINT8 FMSTR_TSA_SINT16 FMSTR_TSA_SINT32 FMSTR_TSA_SINT64	1-, 2-, 4-, or 8-byte signed integer type. Use it for both the standard C-language types (char, short, or long) and the user-defined integer types (Word8, Word16 or Word32, sint8_t, sint16_t, and other).
FMSTR_TSA_FRAC16 FMSTR_TSA_FRAC32 FMSTR_TSA_FRAC64 FMSTR_TSA_FRAC_Q(m,n)	Fractional data types. Although these types are treated as integer types in the C-language, it is beneficial to describe them using these macros so that FreeMASTER treats them properly. Use the Q(m,n) form to describe the general Q fractional number format (m+n+1 = total bits). Supported by FreeMASTER PC Host tool v2.0 (and higher).
FMSTR_TSA_UFRAC16 FMSTR_TSA_UFRAC32 FMSTR_TSA_UFRAC64 FMSTR_TSA_UFRAC_UQ(m,n)	Unsigned fractional data types. Although these types are treated as unsigned integer types in the C-language, it is beneficial to describe them using these macros so that FreeMASTER treats them properly. Use the UQ(m,n) form to describe the general Q fractional number format (m+n = total bits). Supported by FreeMASTER PC Host tool v2.0 (and higher).
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type.
FMSTR_TSA_POINTER	Generic pointer type defined as a 16-bit or 32-bit integer (based on the platform).
FMSTR_TSA_USERTYPE(**name**)	Structure or union type. Specify the type name as the argument.

3.4.2 TSA Table List

There must be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro:

```
FMSTR_TSA_TABLE_LIST_BEGIN()
```

and continues with the TSA table entries for each table:

```
FMSTR_TSA_TABLE(table_id)

FMSTR_TSA_TABLE(table_id2)

FMSTR_TSA_TABLE(table_id3)

...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

3.4.3 TSA Active Content entries

In FreeMASTER v2.0 (and higher), the TSA Active Content is supported, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects in a way similar to accessing the files and folders on the local hard drive.

With this new set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL with the `fmstr:` protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)

        // Directory entry applies to all subsequent MEMFILE entries
        FMSTR_TSA_DIRECTORY("/text_files")          // entering a new virtual directory

        // the readme.txt file will be accessible at URL: “fmstr://text_files/readme.txt”
        FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt))// memory-mapped file

        // files can also be specified with a full path so the DIRECTORY entry does not apply
        FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index))          // memory-mapped file
        FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) // memory-mapped file

        // hyperlinks can point to a local MEMFILE object or to Internet
        FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
        FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

        // project file links simplify opening the projects from any URL
        FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
        FMSTR_TSA_PROJECT("Full Project (online)", “http://mycompany.com/prj/demo.pmp”)

FMSTR_TSA_TABLE_END()
```

3.4.4 FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR nBuffAddr, FMSTR_SIZE nBuffSize);
```

Declaration

freemaster.h

Implementation

freemaster_tsa.c

Arguments

nBuffAddr (in)—address of the memory buffer for the dynamic TSA table

nBuffSize (in)—size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description

This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

3.4.5 FMSTR_TsaAddVar

Prototype

```
FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR pszName, FMSTR_TSATBL_STRPTR pszType,
        FMSTR_TSATBL_VOIDPTR nAddr, FMSTR_SIZE32 nSize, FMSTR_SIZE nFlags);
```

Declaration

freemaster.h

Implementation

freemaster_tsa.c

Arguments

pszName (in)—name of the object

pszType (in)—name of the object type

nAddr (in)—address of the object

nSize (in)—size of the object

nFlags (in)—access flags; a combination of these values:

- FMSTR_TSA_INFO_RO_VAR—read-only memory-mapped object (typically a variable)
- FMSTR_TSA_INFO_RW_VAR—read/write memory-mapped object
- FMSTR_TSA_INFO_STRUCT—heading of the structure or union type definition
- FMSTR_TSA_INFO_MEMBER—member of the structure or union type definition

Description

This function can be called only when the dynamic TSA table is enabled by the FMSTR_USE_TSA_DYNAMIC configuration option and when the FMSTR_SetUpTsaBuff function call is made to assign the dynamic TSA table memory. This function adds one entry into the dynamic TSA table.

It can be used to register a read-only or read/write memory object or describe one item of the user-defined type.

See Section 3.4.1, "TSA table definition" for more details about the TSA table entries.

3.5 Application Commands API

3.5.1 FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_appcmd.c
```

Description

This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the FMSTR_APPCMDRESULT_NOCMD constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the `FMSTR_AppCmdAck` call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The `FMSTR_GetAppCmd` function does not report the commands for which a callback handler function exists. If the `FMSTR_GetAppCmd` function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns FMSTR_APPCMDRESULT_NOCMD.

3.5.2 FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* pDataLen)
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_appcmd.c
```

Arguments

pDataLen (out)—pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description

This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see the `FMSTR_GetAppCmd` function above).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the `FMSTR_AppCmdAck` call, copy the data out to a private buffer.

3.5.3 FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT nResultCode)
```

Declaration

freemaster.h

Implementation

freemaster_appcmd.c

Arguments

nResultCode (in)—the result code which is to be returned to FreeMASTER

Description

This function is used when the Application Command processing finishes in the application. The nResultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

3.5.4 FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(

        FMSTR_ADDR nResultDataAddr,
        FMSTR_SIZE nResultDataLen);
```

Declaration

freemaster.h

Implementation

freemaster_appcmd.c

Arguments

nResultDataAddr (in)—pointer to the data buffer that is to be copied to the Application Command data buffer

nResultDataLen (in)—length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description

This function can be used before the Application Command processing finishes, when there are any data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use the FMSTR_GetAppCmdData and the data buffer after the FMSTR_AppCmdSetResponseData is called.

NOTE

The current version of FreeMASTER does not support the Application Command response data.

3.5.5 FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(
        FMSTR_APPCMD_CODE nAppCmdCode,
        FMSTR_PAPPCMDFUNC pCallbackFunc);
```

Declaration

```
freemaster.h
```

Implementation

```
freemaster_appcmd.c
```

Arguments

nAppCmdCode (in)—Application Command code for which the callback is to be registered

pCallbackFunc (in)—pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return Value

This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when you try to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description

This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using a single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is:

```
FMSTR_APPCMD_RESULT HandlerFunction(
        FMSTR_APPCMD_CODE nAppcmd,
        FMSTR_APPCMD_PDATA pData,
        FMSTR_SIZE nDataLen)
```

Where:

nAppcmd—Application Command code

pData—points to the Application Command data received (if any)

nDataLen—information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

NOTE

The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the `FMSTR_RegisterAppCmdCall` function always fails.

3.6 Pipes API

3.6.1 FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT nPort, FMSTR_PPIPEFUNC pCallback,
        FMSTR_ADDR pRxBuff, FMSTR_PIPE_SIZE nRxSize,
        FMSTR_ADDR pTxBuff, FMSTR_PIPE_SIZE nTxSize);
```

Declaration

`freemaster.h`

Implementation

`freemaster_pipes.c`

Arguments

nPort (in)—port number that identifies the pipe for the client

pCallback (in)—pointer to the callback function which is called whenever a pipe data status changes

pRxBuff (in)—address of the receive memory buffer

nRxSize (in)—size of the receive memory buffer

pTxBuff (in)—address of the transmit memory buffer

nTxSize (in)—size of the transmit memory buffer

Description

This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the `FMSTR_PipeRead` call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the `FMSTR_PipeWrite` call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The ***PipeHandler*** name is only a placeholder and must be defined by the application.

```
void PipeHandler(FMSTR_HPIPE hPipe);
```

3.6.2 FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE hpipe);
```

Declaration

`freemaster.h`

Implementation

`freemaster_pipes.c`

Arguments

hPipe (in)—pipe handle returned from the `FMSTR_PipeOpen` function call

Description

This function deinitializes the pipe object. No data can be received or sent on the pipe after this call.

3.6.3 FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE hPipe, FMSTR_ADDR nAddr,
        FMSTR_PIPE_SIZE nLength, FMSTR_PIPE_SIZE nGranularity);
```

Declaration

`freemaster.h`

Implementation

`freemaster_pipes.c`

Arguments

hPipe (in)—pipe handle returned from the `FMSTR_PipeOpen` function call

nAddr (in)—address of the data to be written

nLength (in)—length of the data to be written

nGranularity (in)—size of the minimum unit of data which is to be written

Description

This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The ***nGranularity*** argument can be used to split the data into smaller chunks, each of the size given by the *nGranularity* value. The `FMSTR_PipeWrite` function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk. This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the *nGranularity* value equal to the *nLength* value, all data are considered as one chunk which is either written successfully as a whole or not at all. The *nGranularity* value of 0 or 1 disables the data-chunk approach.

3.6.4 FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE hPipe, FMSTR_ADDR nAddr,
        FMSTR_PIPE_SIZE nLength, FMSTR_PIPE_SIZE nGranularity);
```

Declaration

`freemaster.h`

Implementation

`freemaster_pipes.c`

Arguments

hPipe (in)—pipe handle returned from the `FMSTR_PipeOpen` function call

nAddr (in)—address of the data buffer to be filled with the received data

nLength (in)—length of the data to be read

nGranularity (in)—size of the minimum unit of data which is to be read

Description

This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The *nGranularity* argument can be used to copy the data in larger chunks in the same way as described in the `FMSTR_PipeWrite` function.

3.7 API Data Types

This section describes the data types used in the FreeMASTER driver. The information provided here can help you to modify or port the FreeMASTER Serial Communication Driver to those NXP platforms that are not officially supported yet.

NOTE

The licensing condition prohibits using FreeMASTER and the FreeMASTER Serial Communication Driver with a non-NXP MPU or MCU products.

The following table describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Table 3-2. Public data types

Type name	Description
FMSTR_ADDR	Data type used to hold the memory address. On most platforms, this is normally a C-pointer, but it may also be a pure integer type. For example, this type is defined as long integer on the 56F8xxx platform where the 24-bit addresses must be supported, but the C-pointer may be only 16 bits wide in some compiler configurations.
FMSTR_SIZE	Data type used to hold the memory block size. It is required that this type is unsigned and at least 16 bits wide integer.
FMSTR_BOOL	Data type used as a general boolean type. This type is used only in zero/non-zero conditions in the driver code.
FMSTR_APPCMD_CODE	Data type used to hold the Application Command code. Generally, this is an unsigned 8-bit value.
FMSTR_APPCMD_DATA	Data type used to create the Application Command data buffer. Generally, this is an unsigned 8-bit value.
FMSTR_APPCMD_RESULT	Data type used to hold the Application Command result code. Generally, this is an unsigned 8-bit value.
FMSTR_PAPPCMDFUNC	Pointer to the Application Command handler function. See `FMSTR_RegisterAppCmdCall` for more details.

The following table describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included to the user application indirectly by the *freemaster.h* file.

Table 3-3. TSA public data types

Type name	Description
FMSTR_TSA_TINDEX	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables. By default, this is defined as FMSTR_SIZE.
FMSTR_TSA_TSIZE	Data type used to hold a memory block size, as used in the TSA descriptors. By default, this is defined as FMSTR_SIZE.

This table describes the data types used by the FreeMASTER Pipes API:

Table 3-4. Pipe-related data types

Type name	Description
FMSTR_HPIPE	Pipe handle which identifies the open-pipe object. Generally, this is a pointer to a void type.
FMSTR_PIPE_PORT	Integer type required to hold at least 7 bits of data. Generally, this is an unsigned 8-bit or 16-bit type.
FMSTR_PIPE_SIZE	Integer type required to hold at least 16 bits of data. This is used to store the data buffer sizes.
FMSTR_PPIPEFUNC	Pointer to the pipe handler function. See `FMSTR_PipeOpen` for more details.

The following table describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and are not available in the application code.

Table 3-5. Private data types

Type name	Description
FMSTR_U8	The smallest memory entity. On the vast majority of platforms, this is an unsigned 8-bit integer. On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.
FMSTR_U16	Unsigned 16-bit integer.
FMSTR_U32	Unsigned 32-bit integer.
FMSTR_S8	Signed 8-bit integer. This type is not defined on the 56F8xx platform.
FMSTR_S16	Signed 16-bit integer.
FMSTR_S32	Signed 32-bit integer.
FMSTR_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_FLAGS	Data type forming a union with a structure of flag bit-fields.
FMSTR_SIZE8	Data type holding a general size value, at least 8 bits wide.
FMSTR_INDEX	General for-loop index. Must be signed, at least 16 bits wide.
FMSTR_BCHR	A single character in the communication buffer. Typically, this is an 8-bit unsigned integer, except for the DSP platforms where it is a 16-bit integer.
FMSTR_BPTR	A pointer to the communication buffer (an array of FMSTR_BCHR).
FMSTR_SCISR	Data type holding the SCI status register value (8, 16, or 32 bits wide, depending on the platform).

Chapter 4 PLATFORM-SPECIFIC TOPICS

The following sections describe the implementation details of each supported platform. See also the *readme.txt* files located in the respective platform-specific subfolders in the *src_platforms* folder.

4.1 Platform-dependent code

Although the FreeMASTER Serial Communication Driver was written in the C language with the cross-platform compatibility and portability in mind, it was necessary to make some small portions of code platform-dependent. These parts of the code are in the platform-dependent part of the driver:

- **Data types**—the code size and performance may be optimized when using the data types natively suitable for each particular platform. For example, the FMSTR_SIZE8 data type is only required to be 8 bits wide in the driver code and it is declared as “unsigned char” on most of the platforms. On the other hand, a better code is generated on the DSP or hybrid controller platforms when this type is declared as a 16-bit integer.
- **Communication buffer access and memory access—**typically, the buffers are simple arrays of bytes on most platforms. However, on the 56F8xx DSP platform, there is nothing like a byte array so a native array of 16-bit integers takes the buffer role.
- **16-bit vs. 32-bit FreeMASTER commands**—to optimize the communication protocol traffic, the FreeMASTER protocol defines the memory accesses of both the 16-bit and 32-bit addresses. The processor platforms differ in the address bus width, so they differ also in how the FreeMASTER driver handles the protocol addresses. On some platforms, it makes sense to implement both modes simultaneously, as there is a kind of “zero-page” RAM which can be addressed using the 16-bit addresses as well as a standard memory requiring 32-bit addressing.
- **SCI /CAN handling**—although the SCI/CAN modules are fairly similar across the NXP platforms, the control and status registers can be organized differently and sometimes need a different handling.
- **Interrupt service routine**—the declaration of the interrupt service routine depends heavily on the platform and on the C compiler used.
- **Fast Recorder implementation**—this is by nature a platform-specific code. Be aware, that the Fast Recorder is supported only on the 56F8xxx platform.

Despite the list of platform-dependent exceptions above, the vast majority of the FreeMASTER driver code is common to all platforms. Rarely, some minor platform or compiler dependencies are solved directly in the shared source code files.

- **Driver initialization and serial line handling**—thanks to the customized SCI access macros for each platform, the general serial communication handler can be written completely platform-independent.
- **Protocol decoder and protocol handlers—**the FreeMASTER protocol is handled completely in the platform-independent part.
- **Oscilloscope and Recorder data sampling**—due to specialized memory access routines (platform-dependent), the rest of the Oscilloscope and Recorder logic is made completely platform-independent.
- **Target-Side Addressing**—all TSA-related code is independent on the target platform.

4.2 56F8xxx digital signal controllers

The digital signal controllers (also referred to as hybrid controllers) of the 56F8300, 56F8100, and 56F8000 families (also known as 56F800E), combine the DSP-optimized instruction set with a standard MCU-like core. Unlike the older 56F8xx family, this platform supports the byte-wise oriented access to the memory.

The most of the porting effort was dedicated to enable the global memory access (24-bit addressing) in the so-called “Small Memory Model” of the CodeWarrior C compiler. In this mode, the compiler assumes the C-pointers are 16 bits wide only, without a concept of the “far” addresses (this concept is to be supported in the future compiler versions). The “address” in the FreeMASTER driver must be allocated and accessed as a standard 32-bit integer, which requires some inline assembly code when accessing the memory.

The 56F8xxx is the only platform that enables the interrupt-driven real-time communication over the EOnCE/JTAG port. The Real-Time Data Exchange (RTDX) feature of the EOnCE port is used in a way similar to how the SCI communicates. The only difference is that the JTAG uses 32-bit data as the basic communication entity, while the SCI uses a standard byte-oriented communication. With JTAG, every four bytes that would be normally sent over the SCI are packed together and transmitted over the JTAG line. Similarly, each double word received from the JTAG port is split into four separate bytes which are fed to the protocol-decoding engine.

4.2.1 56F8xxx-specific driver files

The 56F8xxx-specific code is located in the *\src_platforms\56F8xxx* folder.

- *freemaster.h*—master header file identifying the platform with the FMSTR_PLATFORM_56F8xxx macro constant.
- *freemaster_56F8xxx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - — Using both the 16-bit and 32-bit FreeMASTER protocol commands is enabled by default (FMSTR_USE_NOEX_CMDS and FMSTR_USE_EX_CMDS are both “one” by default)
  - — Inline assembly functions to access the memory with the “non-pointer” addresses.
- *freemaster_56F8xxx.c*—implements the platform-specific memory access functions and code needed to use mixed 16-bit and 32-bit protocol commands.

4.2.2 56F8xxx-specific configuration options

This table describes the configuration options specific to the 56F8xxx platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-1. 56F8xxx platform configuration options

Statement	Values	Description
#define FMSTR_USE_JTAG	boolean (0 or 1)	Define this as non-zero when you want to communicate over the JTAG instead of the SCI. Note that a special JTAG communication plugin for the FreeMASTER tool is required on the PC side.

Table 4-1. 56F8xxx platform configuration options (continued)

Statement	Values	Description
#define FMSTR_USE_JTAG_TXFIX	boolean (0 or 1)	Implements a simple software workaround of the erroneous TDF bit in the 56F8xxx JTAG module. This option must be set according to the FreeMASTER JTAG communication plugin settings.
#define FMSTR_REC_FARBUFF	boolean (0 or 1)	When the recorder is used and the recorder buffer is to be allocated by the driver (FMSTR_REC_OWNBUFF = 0), this option puts the recorder buffer to a section called “fardata”. You can then re-code the linker command file of your project to put the buffer to any arbitrary memory address. If the DSP56F800E_Quick_Start environment is used to develop the application, the linker command files are already set in way that the “fardata” section is put into the external memory, after the address of 0x10000.

4.3 HC08/HCS08 MCUs

Thanks to the simplicity of the instruction set and the powerful CodeWarrior C compiler, only a minimal effort is needed to port the FreeMASTER driver to the HC08 and HCS08 platforms.

4.3.1 HC08-specific driver files

The HC08- and HCS08-specific code is located in the *\src_platforms\HC08* folder.

- *freemaster.h*—this master header file identifies the platform with the FMSTR_PLATFORM_HC08 macro constant.
- *freemaster_HC08.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - — Only the 16-bit FreeMASTER protocol commands are implemented (FMSTR_USE_EX_CMDS is forced to zero).
  - — Because the SCI modules slightly differ on the HC08 and HCS08 families, the SCI access macros are compiled differently in this file (based on the __HCS08__ preprocessor constant).
- *freemaster_HC08.c*—implements the platform-specific memory access functions.

4.3.2 HC08-specific configuration options

This table describes the configuration options specific to the HC08 platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-2. HC08 platform configuration options

Statement	Values	Description
#define FMSTR_SCI_INTERRUPT	vector number	SCI common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.
#define FMSTR_SCI_RX_INTERRUPT	vector number	SCI receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` function is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_SCI_TX_INTERRUPT	vector number	SCI transmit interrupt vector number.
#define FMSTR_CAN_INTERRUPT	vector number	CAN common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.

Table 4-2. HC08 platform configuration options (continued)

Statement	Values	Description
#define FMSTR_CAN_RX_ INTERRUPT	vector number	CAN receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` function is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_CAN_TX_ INTERRUPT	vector number	CAN transmit interrupt vector number.
#define FMSTR_USE_USB_CDC	boolean (0 or 1)	Define it as non-zero when you want to communicate over the USB interface instead of the SCI, JTAG, or CAN. Note that this option requires Medical USB stack v3.1 (or higher) to be included into your project with the USB CDC class support.

4.4 HC12/HCS12/HCS12X/S12Z MCUs

Supporting only the 16-bit data addressing, the HC12 and HCS12 driver code is very similar to the HC08 code. The 24-bit PPAGE-based “logical” addresses do not have a native support on these platforms, so this addressing mode is NOT supported in the FreeMASTER driver.

The HCS12X and the new GPAGE-based “global” addressing modes enable transparent access to the paged memory, so it is supported also by the FreeMASTER driver. The HCS12X version of the driver also performs an automatic translation from the PPAGE- and RPAGE-based logical addresses to the global addresses. See the FMSTR_LARGE_MODEL configuration option in Table 4-1.

4.4.1 HC12-specific driver files

The HC12-, HCS12-, and HCS12X-specific code can be found in the *\src_platforms\HC12* folder.

- *freemaster.h*—this master header file identifies the platform with the FMSTR_PLATFORM_HC12 macro constant.
- *freemaster_HC12.h*—this file contains the driver options specific to this platform, and several other platform-specific memory access inline functions and macros.
  - For the HC12 and HCS12 devices, only 16-bit FreeMASTER protocol commands are supported (FMSTR_USE_EX_CMDS is forced to zero).
  - For HCS12X, the 16-bit commands are enabled by default. The 32-bit commands are enabled in the “large” data memory models only. Both the HCS12X global and logical addresses are correctly handled by the driver.
- *freemaster_HC12.c*—this file implements the platform-specific memory access functions. The HCS12X large model addressing, logical-to-global address translation, and other HCS12X-specific code is also implemented in this file.

4.4.2 HC12-specific configuration options

This table describes the configuration options specific to the HC12 platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-3. HC12 platform configuration options

Statement	Values	Description
#define FMSTR_LARGE_MODEL	boolean (0 or 1)	**HCS12X only**: Enable the support for 24-bit “large” addressing. When enabled, the serial driver properly decodes all kinds of memory addresses: - 16-bit “near” addresses - 23-bit “global” addresses - 24-bit “logical” addresses (PAGE + offset) Default: “false” in the SMALL or BANKED compiler modes Default: “true” in the LARGE compiler mode
#define FMSTR_SCI_INTERRUPT	vector number	SCI common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.
#define FMSTR_SCI_RX_INTERRUPT	vector number	SCI receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` function is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_SCI_TX_INTERRUPT	vector number	SCI transmit interrupt vector number.
#define FMSTR_CAN_INTERRUPT	vector number	CAN common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.
#define FMSTR_CAN_RX_ INTERRUPT	vector number	CAN receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` function is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_CAN_TX_ INTERRUPT	vector number	CAN transmit interrupt vector number.
#define FMSTR_USE_LINTL	boolean (0 or 1)	**S12Z platforms only:** Define it as non-zero when you want to communicate over the LIN Transport Layer protocol API. The LIN driver with the standard Transport Layer API must be compiled in the application too. Note that a special LIN communication plugin for FreeMASTER is required on the PC side. Default: “false”

4.5 MPC55xx Power Architecture MCUs

The Power Architecture is a true 32-bit platform with a powerful CodeWarrior C compiler.

4.5.1 MPC55xx-specific driver files

The MPC55xx-specific code is located in the *\src_platforms\MPC55xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MPC55xx macro constant.
- *freemaster_MPC55xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.

  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_MPC55xx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.5.2 MPC55xx-specific configuration options

None. Only the standard options are used, as described in Section 2.4, "Driver configuration".

4.6 MPC56xx Power Architecture MCUs

The Power Architecture is a true 32-bit platform with a powerful CodeWarrior Eclipse IDE.

4.6.1 MPC56xx-specific driver files

The MPC56xx-specific code is located in the *\src_platforms\MPC56xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MPC56xx macro constant.
- *freemaster_MPC56xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_MPC56xx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.6.2 MPC56xx-specific configuration options

None. Only the standard options are used, as described in Section 2.4, "Driver configuration".

NOTE

The data cache must be disabled when the BDM or PBDM communication plugins are used. The BDM interface accesses the physical memory while the CPU accesses the data cache. This makes the BDM and PDBDM plugins unusable for the FreeMASTER communication. This assumption is valid only for the MPC56xx platforms whose cores contain data cache.

4.7 MPC57xx Power Architecture MCUs

The Power Architecture is a true 32-bit platform with the powerful S32 Design Studio for Power Architecture IDE.

4.7.1 MPC57xx-specific driver files

The MPC57xx-specific code can be found in the *\src_platforms\MPC57xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MPC57xx macro constant.
- *freemaster_MPC57xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.

  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_MPC57xx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.7.2 MPC57xx-specific configuration options

None. Only the standard options are used, as described in Section 2.4, "Driver configuration".

NOTE

The data cache must be disabled when the BDM or PBDM communication plugins are used. The BDM interface accesses the physical memory while the CPU accesses the data cache. This makes the BDM and PDBDM plugins unusable for the FreeMASTER communication.

4.8 KEAxx ARM Cortex-M MCUs

The KEAxx is a 32-bit MCU family based on the ARM Cortex-M0+ architecture. This platform is fully supported by FreeMASTER with the powerful S32 Design Studio for ARM IDE.

4.8.1 KEAxx-specific driver files

The S32xx-specific code is located in the *\src_platforms\KEAxx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_KEAxx macro constant.
- *freemaster_KEAxx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to boolean 1). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to boolean 0).
- *freemaster_KEAxx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.8.2 KEAxx-specific configuration options

None. Only the standard options are used, as described in Section 2.4, "Driver configuration".

NOTE

The Cortex-M0 devices are not capable of misaligned memory access, so the FreeMASTER driver must be configured for FMSTR_BYTE_BUFFER_ACCESS when used with the Cortex-M0+ devices.

4.9 S32xx ARM Cortex-M MCUs

The S32xx is a 32-bit MCU family based on the ARM Cortex architecture. This platform is fully supported by FreeMASTER with the powerful S32 Design Studio for ARM IDE.

4.9.1 S32xx-specific driver files

The S32xx-specific code is located in the *\src_platforms\S32xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_S32xx macro constant.
- *freemaster_S32xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_S32xx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.9.2 S32xx-specific configuration options

None. Only the standard options are used, as described in Section 2.4, "Driver configuration".

4.10 MCF51xx ColdFire processors

Similarly to the HC08 processors, the FreeMASTER driver is ported to the ColdFire V1 platform with only a minimum effort.

4.10.1 MCF51xx-specific driver files

The MCF51xx-specific code is located in the *\src_platforms\MCF51xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MCF51xx macro constant.
- *freemaster_MCF51xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_MCF51xx.c*—implements the platform-specific memory access functions and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.10.2 MCF51xx-specific configuration options

This table describes the configuration options specific to the MCF51xx platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-4. MCF51xx platform configuration options

Statement	Values	Description
#define FMSTR_SCI_INTERRUPT	vector number	SCI common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.
#define FMSTR_SCI_RX_INTERRUPT	vector number	SCI receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_SCI_TX_INTERRUPT	vector number	SCI transmit interrupt vector number.

Table 4-4. MCF51xx platform configuration options (continued)

Statement	Values	Description
#define FMSTR_CAN_INTERRUPT	vector number	CAN common interrupt vector number. Define this number only when both the RX and TX interrupt vectors share one interrupt vector number. In this case, the `FMSTR_Isr` function is installed automatically as the interrupt service routine.
#define FMSTR_CAN_RX_ INTERRUPT	vector number	CAN receive interrupt vector number. When both the RX and TX interrupt vector numbers are defined, the `FMSTR_Isr` function is installed automatically as the interrupt service routine for both interrupts (the ”interrupt” keyword is used for the compiler).
#define FMSTR_CAN_TX_ INTERRUPT	vector number	CAN transmit interrupt vector number.
#define FMSTR_USE_USB_CDC	boolean (0 or 1)	Define it as non-zero when you want to communicate over the USB interface instead of the SCI, JTAG, or CAN. Note that this option requires Medical USB stack v3.1 (or higher) to be included into your project with the USB CDC class support.

4.11 MCF52xx ColdFire processors

Similarly to the Power Architecture processors, the FreeMASTER driver is ported to the ColdFire platform with only a minimum effort.

4.11.1 MCF52xx-specific driver files

The MCF52xx-specific code is located in the *\src_platforms\MCF52xx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MCF52xx macro constant.
- *freemaster_MCF52xx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_MCF52xx.c*—implements the platform-specific memory access functions, and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.11.2 MCF52xx-specific configuration options

This table describes the configuration options specific to the MCF52xx platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-5. MCF52xx platform configuration options

Statement	Values	Description
#define FMSTR_USE_USB_CDC	boolean (0 or 1)	Define it as non-zero when you want to communicate over the USB interface instead of the SCI, JTAG, or CAN. Note that this option requires Medical USB stack v3.1 (or higher) to be included into your project with the USB CDC class support.

4.12 Kxx Kinetis ARM Cortex-M MCUs

Kinetis is a 32-bit MCU family based on the ARM Cortex-M0+ or Cortex-M4 architecture. This platform is fully supported by FreeMASTER.

4.12.1 Kxx-specific driver files

The Kxx-specific code is located in the *\src_platforms\Kxx* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_Kxx macro constant.
- *freemaster_Kxx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_Kxx.c*—implements the platform-specific memory access functions, and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.12.2 Kxx-specific configuration options

This table describes the configuration options specific to the Kxx platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-6. Kxx platform configuration options

Statement	Values	Description
#define FMSTR_USE_USB_CDC	boolean (0 or 1)	Define it as non-zero when you want to communicate over the USB interface instead of the SCI, JTAG, or CAN. Note that this option requires Medical USB stack v3.1 (or higher) to be included into your project with the USB CDC class support.
#define FMSTR_USE_MBED	boolean (0 or 1)	Define it as non-zero when the driver is used with the serial_t object defined by the ARM mbed development environment.

NOTE

The Cortex-M0 devices are not capable of a misaligned memory access, so the FreeMASTER driver must be configured for FMSTR_BYTE_BUFFER_ACCESS when used with the Cortex-M0+ devices.

4.13 MQX RTOS operating system platform

MQX is a real-time operating system, which supports several platforms (MCF51xx, MCF52xx, Kxx, and others).

4.13.1 MQX-specific driver files

The MQX-specific code is located in the *\src_platforms\MQX* folder.

- *freemaster.h*—master header file which identifies the platform with the FMSTR_PLATFORM_MQX macro constant.

- *freemaster_mqx.h*—contains the driver options specific to this platform and several other platform-specific memory access inline functions and macros.
  - The 32-bit FreeMASTER protocol commands are enabled by default (FMSTR_USE_EX_CMDS defaults to one). The support for 16-bit commands is possible, but it is not enabled by default (FMSTR_USE_NOEX_CMDS defaults to zero).
- *freemaster_mqx.c*—implements the platform-specific memory access functions, and handles the mixed 16-bit and 32-bit protocol commands (if both are used).

4.13.2 MQX-specific configuration options

This table describes the configuration options specific to the MQX platform, used in addition to those described in Section 2.4, "Driver configuration":

Table 4-7. MQX platform configuration options

Statement	Values	Description
#define FMSTR_USE_MQX_IO	boolean (0 or 1)	Define it as non-zero when you want to communicate over the MQX standard IO API function. Default is “false”.
#define FMSTR_MQX_IO_CHANNEL	string (“ttya:“, “ittya“)	Define the name of the MQX interface to by used as the FreeMASTER communication interface.
#define FMSTR_MQX_IO_BLOCKING	boolean (0 or 1)	Define it as non-zero when you want to open the MQX IO interface in the non-blocking mode—the `FMSTR_Poll()` function is waiting until a command from the PC side is received. Default is “false”.

Appendix A References

- *DSP56F800 User Manual* (document DSP56F801-7UM)
- *56F8300 Peripheral User Manual* (document MC56F8300UM)
- *56F80xx Peripheral User Manual* (document MC56F8000RM)
- *CPU08 Central Processing Unit* (document CPU08RM)
- *M68HC12 & HCS12 Microcontrollers* (document CPU12RM)
- *MPC5553/5554 Microcontroller Reference Manual* (document MPC5553/4RM)
- *MPC555/556 User's Manual and Data Sheet* (document MPC555UM)
- *MPC5604P Microcontroller Reference Manual* (document MPC560xPRM)
- *MPC5604B/C Microcontroller Reference Manual* (document MPC5604BCRM)
- *MPC5634M Microcontroller Reference Manual* (document MPC563XMRM)
- *MPC5604E Microcontroller Reference Manual* (document MPC5604ERM)
- *MPC5606S Microcontroller Reference Manual* (document MPC5606SRM)
- *MPC5644A Microcontroller Reference Manual* (document MPC5644ARM)
- *MPC5643L Microcontroller Reference Manual* (document MPC5643LRM)
- *MPC5676R Microcontroller Reference Manual* (document MPC5676RRM)
- *MPC5675K Microcontroller Reference Manual* (document MPC5675KRM)
- *MPC5674F Microcontroller Reference Manual* (document MPC5674FRM)
- *MPC5744P Microcontroller Reference Manual* (document MPC5744PRM)
- *MPC5777C Microcontroller Reference Manual* (document MPC5777CRM)
- *MPC5746C Microcontroller Reference Manual* (document MPC5746CRM)
- *MPC5748G Microcontroller Reference Manual* (document MPC5748GRM)
- *MPC5777M Microcontroller Reference Manual* (document MPC5777MRM)
- *MPC5746R Microcontroller Reference Manual* (document MPC5746RRM)
- *KEA128 Microcontroller Reference Manual* (document KEA128RM)
- *S32K14 Microcontroller Reference Manual* (document S32K14XRM)
- *MCF51QE128 Microcontroller Reference Manual* (document MCF51QE128RM)
- *MC9S12ZVM-Family Reference Manual and Data Sheet* (document MC9S12ZVMRM)
- *K60 Sub-Family Reference Manual* (document K60P121M100SF2RM)
- *NXP MQX TM I/O Drivers* (document MQXIOUG)
- *FreeMASTER Usage Serial Driver Implementation* (document AN4752)
- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document AN4771)
- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document AN4860)

Appendix B Revision History

This table summarizes the changes done to this document since the initial release:

Table 4-8. Revision history

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. Added MCAN interface. Folders structure at the installation destination was rearranged.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: FMSTERSCIDRVUG
Rev. 3.0
08/2016